Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Design und Implementierung eines optimierenden VHBC-Compilers für die Virtual Hardware Machine
- und -
Realisierung der Virtual Hardware Machine in VHDL

Diplomarbeit

Leipzig, 13. Juli 2006

geschrieben von
Thomas Siegmund

Zusammenfassung

Die vorliegende Arbeit beschreibt, auf den Ergebnissen von Sebastian Lange [1] aufbauend, die Optimierung des VHBC-Compilers, die Erweiterung der Eingabedateiformate des Compilers um EDIF-Netzlisten, seine Anpassung an die veränderte Architektur der VHM und die Realisierung dieser Architektur mittels VHDL, basierend auf der Diplomarbeit von Nick Bierwisch[2], welcher in seiner Arbeit die erste Implementierung der VHM schuf.

Es wird der Aufbau und die Arbeitsweise des VHBC-Compilers erläutert und die neue Architektur der VHM ausführlich beschrieben. Dem geht ein Vergleich mit bestehenden Ansätzen rekonfigurierbarer Hardware und eine Analyse der Schwachpunkte der bestehenden VHM und des VHBC-Compilers voraus.

Danksagung

Ich möchte mich recht herzlich bei Prof. Dr. Udo Kebschull dafür bedanken, dass er mir dieses äußerst interessante Thema zur Bearbeitung überlassen hat. Danke für die Geduld die Sie für mich aufgebracht haben und die Unterstützung während der ganzen Zeit.

Weiterhin danke ich Sebastian Lange, der immer da war wenn, man Hilfe brauchte und mir mit konstruktiven Vorschlägen weiterhelfen konnte.

Besonderer Dank gebührt meinen Eltern, welche mich so viele Jahre ausgehalten haben und immer für mich da waren. Ich schulde euch was.

Zum Schluss möchte ich noch Matthias Bernt, Michael Barth und Klaus ’Ekki’ Fischer danken, welche ihre kostbare Zeit mit Korrektur lesen verbracht haben und ausdauernd meinem Gerede über die VHM zuhörten. Danke.

Danke auch an alle die mir geholfen haben und die ich hier jetzt nicht namentlich erwähnt habe.

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
EDIF	Electronic Design Interchange Format
EPROM	Electrically Programmable Read Only Memory
EEPROM	Electrically Eraseable Programmable Read Only Memory
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
LUT	Look-Up-Table
MIMD	Multiple Instruction Multiple Data
RPU	Reconfigurable Processing Unit
RT	Register Transfer; in Verbindung mit RT-Ebene
SIMD	Single Instruction Multiple Data
VHBC	Virtual Hardware Byte Code
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VHM	Virtual Hardware Machine
VLIW	Very Long Instuction Word

Inhaltsverzeichnis

1 Einleitung/Motivation **7**

1.1 Der Trend weg vom Statischen 7

1.2 Die Idee hinter der VHM 8

1.3 Aufgabenstellung 9

2 Grundlagen **10**

2.1 Electronic Design Interchange Format (EDIF) 10

2.2 Rekonfigurierbare Hardware 14

2.3 State of the art - vergleichbare Ansätze 16

2.3.1 PACT XPP 16

2.3.2 Quicksilver 21

2.3.3 MorphoSys 26

2.4 VHM und VHBC - Vorausgegangene Arbeiten 30

3 Ansatz **34**

3.1 Die Schwachpunkte der alten VHM 34

3.1.1 Lösungsansätze 35

3.2 Die Schwachpunkte des VHBC-Compilers 38

3.2.1 Lösungsansätze 38

3.3 Zusammenfassung 40

4 Implementierung **41**

4.1 Die Virtuelle Hardware Machine 43

4.1.1 Logikebene 44

4.1.2 RT-Ebene 49

4.1.3 VHM 53

4.2 VHBC-Compiler 57

4.2.1 Phase 1: Eingabe 57

4.2.2 Phase 2: Code-Manipulation 58
4.2.3 Phase 3: Ausgabe 61
4.2.4 Anmerkung zu den Konstanten 64

5 Ergebnisse **66**
5.1 Ressourcenverbrauch und Geschwindigkeit der VHM 66
5.2 Messungen 66
5.2.1 3x3 Medianfilter (Anwendung in der Bildverarbeitung) 67
5.2.2 Farbraumkonverter (RGB ->Y Pr Pb) 68
5.3 Flächenvergleich 69
5.4 Konfigurationsaufwand 69
5.5 Flexibilität 70
5.6 Fazit 70

6 Zusammenfassung und Ausblick **71**
6.1 Was wurde erreicht? 71
6.2 Zukünftige Arbeiten und Verbesserungsvorschläge 72

7 Verwendung der Software **73**
7.1 VHBC-Compiler 73

A CD **76**

Literaturverzeichnis **77**

Kapitel 1

Einleitung/Motivation

1.1 Der Trend weg vom Statischen

Hardware war immer etwas Statisches. Schaltungen wurden für ein Aufgabengebiet entwickelt. Änderten sich die Bedingungen oder das Anwendungsgebiet, so wurde eine bestehende Schaltung angepasst oder eine neue entwickelt. Es handelt sich bei diesen Chips z.B. um ASICs oder GPPs. Diese beiden stellen jeweils ein Ende des Spektrums dar. ASICs sind klein, schnell, energiesparend und günstig in der Massenproduktion, doch eingeschränkt durch ihre Eigenschaft nur für eine ganz spezielle Aufgabe entworfen zu sein. GPPs sind, was Größe, Geschwindigkeit und Energieverbrauch angeht, genau das Gegenteil von ASICs. Ihr großer Vorteil liegt in ihrer Flexibilität, die durch einen Befehlssatz erreicht wird, welcher eine GPP in die Lage versetzt mit Hilfe von Software jedes berechenbare Problem zu lösen. So könnte man meinen, dass durch diese beiden Extreme der ganze Bereich, der durch Hardware zu realisieren ist, abgedeckt wird.
Doch nach und nach änderten sich die Bedingungen. Die Lebenszyklen der produzierten Chips wurden immer kürzer und die Bandbreite der zur Verfügung zu stellenden Operationen immer größer, so dass die immer neue Produktion von Chips in keinem akzeptablen Kosten/Nutzen-Verhältnis mehr stand. Wollte man nun aber den immer größer werdenden Funktionsumfang mittels GPPs realisieren, so hatte man das Problem, dass sie zu viel Energie verbrauchten, was z.B. im Mobile-Bereich einen sehr kritischer Faktor darstellt.

Einen Ausweg aus diesem Dilemma ist sogenannte rekonfigurierbare Hardware. Bei dieser handelt es sich um Chips, welche aus einer Menge von Zellen bestehen, wobei eine Zelle ein logischer Block mit konfigurierbarer Funktionalität ist. Die Zellen sind durch ein konfigurierbares Netzwerk miteinander verbunden. Da die Zellen in ihrem Verhalten nicht festgelegt sind, ermöglichen sie somit dem Benutzer die Realisierung jeder beliebigen Schaltung, sofern die Anzahl der Zellen ausreichend ist.

Durch rekonfigurierbare Hardware ist es also möglich geworden, den teuren und langen Entwicklungszyklus von Siliziumchips kostensparender und kürzer zu gestalten. Dies wird durch mehrere Dinge erzielt. Zum einen können die Entwickler die von ihnen entworfene Schaltung direkt in Hardware testen und müssen somit nicht auf die wesentlich langsamere Simulation mittels Soft-

ware zurück greifen oder die Schaltung in Silizium realisieren, was mit erheblichen Kosten verbunden ist. Weiterhin besteht die Möglichkeit der Wiederverwendung vorhandener Hardware, da nicht mehr zwingend ein Chip in einer Schaltung ausgetauscht werden muss, sondern nur noch seine Konfiguration.

Somit sind Hersteller heutzutage in der Lage die Funktionalität von Geräten zu verändern, ohne auch nur eine Änderung an der Hardware des Geräts vornehmen zu müssen. Das all diese Eigenschaften auch dem Benutzer zugute kommen ist klar, denn die gewonnene Flexibilität in den Geräten erhöht auch seine Flexibilität und spart ihm letzten Endes ebenfalls Geld. Einige Hersteller gehen auch den Weg, dass sie die erste Generation eines Geräts mit Hilfe rekonfigurierbarer Hardware realisieren, um somit ihr Produkt schnellen auf den Markt bringen zu können. Die folgenden Generationen des selben Produkts basiert dann auf fester Hardware. Durch dieses Verhalten verschaffen sie sich den notwendigen Vorsprung vor der Konkurrenz. Das die auf rekonfigurierbarer Hardware basierende Lösung etwas langsamer, größer und leistungshungriger ist als die späteren, ASIC-basierten, wird in diesem Fall in Kauf genommen.

1.2 Die Idee hinter der VHM

Sebastian Lange schreibt in seiner Diplomarbeit, dass die Idee hinter dem VHM-Ansatz es ist, ein Konzept für Hardware zu entwerfen, ähnlich dem Java-Konzept im Softwarebereich, welches die Kompatibilitätsprobleme zwischen den rekonfigurierbaren Architekturen beseitigt. Der VHM-Ansatz bedient sich hierfür zweier Dinge, einer abstrakten Hardwarebeschreibung (VHBC) und eines Hardwareprozessors (VHM), der nicht zwingend mit Hilfe rekonfigurierbarer Hardware realisiert werden muss, sondern auch als ASIC implementiert werden kann.

Der VHBC beschreibt eine Schaltung und stellt diese mit Hilfe einer Menge von festgelegten Instruktionen dar, welche auf einer Menge von Registern operieren. Aufgrund dieses, auf Instruktionen basierenden Aufbaus kann der VHBC nur synchrone Schaltungen repräsentieren. Dabei wird ein Taktzyklus der repräsentierten Schaltung in kleinere Unterzyklen zerlegt, wobei ein Unterzyklus einer Ebene von Instruktionen entspricht, die parallel ausgeführt werden können. Abbildung 1 verdeutlicht die Überführung von Hardware in VHBC.

In Anlehnung an die Java Virtual Machine bietet die VHM die Möglichkeit einer rekonfigurierbaren und platformunabhängigen Hardware. Die VHM ist genauer gesagt ein Prozessor, welcher auf die Ausführung von Hardwaredesigns spezialisiert wurde. Sie soll von der zugrunde liegenden Hardware abstrahieren und es ermöglichen, das Verhalten einer einmal synthetisierten Schaltung zu übernehmen. Dabei spielt es keine Rolle, ob die VHM mit Hilfe von statischer oder rekonfigurierbarer Hardware realisiert wurde. Der Grundgedanke hinter der VHM ist so gesehen dem von JAVA[16] sehr ähnlich, nur daß die VHM diesen Gedanken auf der Basis von Hardware verwirklicht. Dabei erzeugt der VHBC-Compiler aus Netzlisten einen von der zugrunde liegenden Hardware abstrahierenden Bytecode, welcher von der VHM ausgeführt werden kann. Somit ist bloß die Synthese der VHM und nicht für jede zur Implementierung anstehenden Schaltung notwendig. Diese Schaltungen müssen lediglich mit dem VHBC-Compiler übersetzt werden. Durch

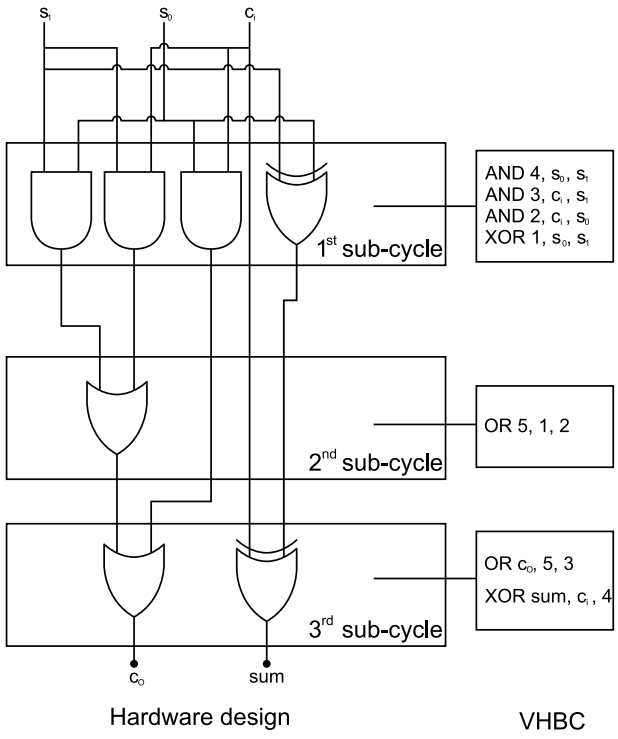


Abbildung 1.1: reale Hardware und ihre Repräsentation in VHBC[1]

diesen Entwicklungsfluss ist es möglich, die Flexibilität von Softwaredesigns auch im Bereich der Hardware einzusetzen. Dieser Ansatz ermöglicht es, fehlerhafte Schaltungen ohne Rekonfiguration (im Sinne von FPGA-Rekonfigurierung) oder Austausch zu reparieren. Desweiteren könnte dieselbe VHM für verschiedene Schaltungen gleichzeitig verwendet werden, indem man, ähnlich dem Taskswitching in Betriebssystemen, die momentane Konfiguration im Speicher ablegt und eine Neue lädt.

1.3 Aufgabenstellung

Das Ziel der vorliegenden Diplomarbeit ist zum einen die Evaluierung des bestehenden Bytecodes und Compilers und die Erarbeitung und Implementierung von Optimierungsstrategien. Das zweite Ziel ist die Realisierung der VHM mit Hilfe von VHDL. Es soll im Ergebnis eine lauf- und synthesefähige VHDL-Beschreibung der VHM erstellt werden, welche den vom Compiler erzeugten Bytecode einlesen und verarbeiten kann. Ferner sollen die Möglichkeiten der Erweiterbarkeit des Bytecodes und der VHM in Betracht gezogen und diese eventuell modifiziert werden.

Kapitel 2

Grundlagen

Dieses Kapitel gibt eine Übersicht über die Grundlagen, auf welchen diese Arbeit aufbaut. Angefangen mit einer allgemeinen Einführung in rekonfigurierbare Hardware und ihrer Relevanz in heutigen Hardwaresystemen und einer Einführung in EDIF, werden in den darauf folgenden Abschnitten verschiedene Implementierungen von zur Laufzeit rekonfigurierbaren Systemen vorgestellt. Abgeschlossen wird dieses Kapitel mit einer kurzen Beschreibung bereits vorausgegangener Arbeiten auf diesem Gebiet.

2.1 Electronic Design Interchange Format (EDIF)

Anfang der der 80er Jahre wurden für die Entwicklung von Hardware Softwaretools verwendet, welche zumeist hochgradig proprietäre Design-Datenbanken verwendeten. Dieser Umstand machte es den Entwicklern sehr schwer bzw. unmöglich ihr Design zwischen zwei verschiedenen Entwicklungsumgebungen auszutauschen. Es existierten zwar Konverter die in der Lage waren diesen Austausch zu ermöglichen, doch ergab sich für diese, aufgrund der steigenden Anzahl verschiedener Formate, ein immer größeres Problem alle Formate abzudecken. Um dem entgegenzuwirken, begann man wurde ein neutrales Format für den Austausch von Designdaten elektronischer Schaltungen auszuarbeitet. Aus diesen Bemühungen heraus entstand 1983 das EDIF Steering Committee, welches aus Vertretern von Daisy Systems, Mentor Graphics, Motorola, National Semiconductor, Tektronix, Texas Instruments und der Univerity of California, Berkeley bestand. Dieses erstellte bis 1985 die EDIF-Version 1 0 0, welcher aber keine größere Bedeutung zukam. Die erste praxisrelevante Veröffentlichung stellte EDIF 2 0 0 im Jahre 1987 dar, welches im Folgejahr ANSI[1] uns EIA[2] Standard wurde. Darauf folgten 1993 EDIF 3 0 0[3] und 1996 EDIF 4 0 0. Nach der Veröffentlichung der letzten Version hat sich die EDIF-Standard-Organisation aufgelöst.

Anfangs als CAD-to-CAD[4] Datentransferformat angelegt, wurde EDIF nach und nach so erwei-

[1] American National Standards Institute
[2] Electronic Industries Alliance (Standard EIA-548)
[3] EIA-618 Standard
[4] Computer Aided Design

tert, dass es auch als CAD-to-CAM[5] Transferformat dienen kann. Einer der größten Vorteile des EDIF ist aber, dass es die verlustfreie Darstellung des kompletten Designs in einer einzigen Datei erlaubt. So kann eine EDIF-Datei sowohl Schaltbild, Netzliste, Verhaltensbeschreibung, Dokumentation, Maskenlayout als auch Simulationsmodelle einer Schaltung enthalten.

Die am weitesten verbreitete EDIF Version ist 2 0 0. Sie deckte alle Bereiche elektronischer Designs ab und wurde zum Austausch von Designdaten seit 1987 erfolgreich eingesetzt. Dennoch bekam diese Version einen schlechten Ruf als unzuverlässiges und nicht effektives Austauschformat, was auf einige massive Mängel in der Syntax- und der Semantikdefinition zurückzuführen ist. EDIF 2 0 0 stellte zwar ein einfaches Dateiformat dar, doch implementierten die meisten Firmen nur Importfilter, was sich erst auf Drängen einiger großer Firmen änderte. Die von den Herstellern daraufhin implementierten Exportfilter waren aber zumeist nicht standardkonform, da es aufgrund der Mängel in EDIF 2 0 0 mehrere Wege zu Datendeklaration (speziell Busse) gibt und beinahe jeder Hersteller einen anderen Weg ging. Ein weiteres Problem beim Austausch von Designdaten war eine gewisse Inkompatibilität der Konzepte der einzelnen EDAs[6], was zwangsläufig zu verschiedenen Beschreibungen der gleichen Sache führte. Inkompatibilitäten sind z.B verschiedene Verfahren zur Benennung von Zellen, Instanzen, Eigenschaften, etc oder die unterschiedliche Handhabung von Eigenschaften. Einige spezielle Mängel des EDIF 2 0 0 sind z.B.

- Repräsentation von Bussen ist nicht eindeutig und unvollständig
- Ripper sind schwer zu verstehen und zu benutzen
- eine VIEW-weite Anschlussmöglichkeit wurde für Schaltbilder nicht spezifiziert
- einige Objekte sind einfach nicht implementiert (z.B. Frames)

EDIF 3 0 0 deckt im Gegensatz zu EDIF 2 0 0 nur die Bereiche Schaltbilder und Verbindungen (Busse, Ripper, ...) ab, weist aber in diesen nahezu keine Probleme der Vorgängerversion auf. Nach seiner Veröffentlichung 1993 wurde es ebenfalls internationaler Standard (ANSI/EIA/IEC[7]). Es ist nicht abwärtskompatibel zu EDIF 2 0 0 und erforderte vollkommen neue Import- uns Exportfilter für die Entwicklungswerkzeuge. Während durch die Syntaxverschärfung in EDIF 3 0 0, welche durchgeführt wurde, um die Zweideutigkeiten aus EDIF 2 0 0 zu beseitigen, die Implementierung von Importfiltern sehr einfach wurde, gestaltete sich die Entwicklung von Exportfilter nun wesentlich schwerer. Die logische Konsequenz und die, für die Verwendbarkeit von EDIF beste Lösung war, die Entwicklung der In- und Exportfilter Drittanbietern zu überlassen, welche auf Schnittstellen der Entwicklungswerkzeuge aufsetzen und aufgrund der Unabhängigkeit von Herstellen, welche Entwicklungswerkzeuge vertreiben, standardkonforme Filter realisieren. Das hat zum Ergebnis, dass heutzutage so gut wie kein Hersteller von Entwicklungswerkzeugen mehr eigene EDIF-Werkzeuge implementiert, sonder auf die der Drittanbieter zurückgreift.

[5] Computer Aided Manufacturing

[6] Electronic Design Automation (häufig auch als ECAD bezeichnet); Sammelbegriff für Programme, mit denen elektronische Schaltungen entworfen werden

[7] International Electrotechnical Commission

Einer der größten Vorteile von EDIF 3 0 0 war, dass infolge der Syntaxverschärfung ein automatisches Überprüfen der Syntax und der Semantik möglich war und somit schnell und zuverlässig überprüft werden kann, ob es sich um ein gültiges EDIF-Dokument handelt oder nicht.

Bei EDIF 4 0 0 handelt es sich im Prinzip um einer Erweiterung von EDIF 3 0 0 um PCBs[8] und MCMs[9]. Durch das ECCE[10] Projekt wurden zu EDIF 4 0 0 noch einige weitere Konzepte hinzugefügt, um CAD-to-CAM Datentransfers zu ermöglichen.

EDIF ist ein vom Menschen lesbares und von Maschinen leicht zu parsendes Format. Aufgrund seiner Prefixnotation ähnelt es LISP. Jede EDIF-Anweisung besteht aus einer öffnenden Klammer, einem Schlüsselwort, einer Menge von Parametern und einer schließenden Klammer. Seine Struktur erhält das EDIF dadurch, dass die Parameter einer EDIF-Anweisung wiederum EDIF-Anweisungen seien können. Eine EDIF-Datei besteht sogesehen aus einer einzigen Anweisung: (edif *parameters*), wobei *parameters* die Schaltung beschreibt. Im Interesse des leichten Parsens gibt es drei Level im EDIF, wobei mit steigendem Level ebenfalls die Mächtigkeit steigt. So enthält beispielsweise Level 1, auch *intermediate level* genannt, die Möglichkeit der Verwendung von Variablen und der Verwendung von Parametern in Zellendefinitionen, während Level 0 nicht variabelt ist und ausschließlich auf konstanten Werten basiert. Die Ähnlichkeit zu LISP wird besonders dadurch deutlich, dass EDIF im 3-ten und höchsten Level alles bietet, was LISP zu bieten hat und somit eine Erweiterung dieser darstellt. Einem LISP-Preprozessor ist es dadurch möglich, EDIF Level 1 und 2 in Level 0 zu übersetzten, welches alles Notwendige für den Austausch und alle Einzelheiten für die Herstellung von Schaltungen enthält.

```
(edif name
    (status information)
    (design where-to-find-them)
    (external reference-library)
    (library name
        (technology defaults)
        (cell name
            (viewmap map)
            (view type name
                (interface external)
                (contents internal)
            )
        )
    )
)
```

Abbildung 2.1: Allgemeine Struktur des EDIF

Der Aufbau einer EDIF-Datei ist recht einfach (Abb. 2.1). Die Datei enthält eine Menge von *libraries*, welche wiederum eine Menge von *cells* enthalten. Jede *cell* kann mit Hilfe von einer oder mehreren *views* beschrieben werden, wobei diese die *cell* in einer bestimmten Form, z.B. *schematic* (Schaltbild), *layout*, *behavioral specification* (Verhaltensbeschreibung) oder *documentation*, zeigen. Jede *view* beinhaltet einen *interface* und einen *contents* Teil und kann durch eine *viewmap* Anweisung mit anderen *views* verbunden werden. In *libraries* können auch noch *techno-*

[8] Printed Circuit Board
[9] Multi-Chip Module
[10] Electronic CAD-CAM Exchange

logy Informationen enthalten sein, um Voreinstellungen für Verhalten, Grafik oder andere Attribute festzulegen. Ein konkretes Beispiel eines AND ist in Abbildung 2.2 dargestellt. Eine detaillierte Beschreibung der EDIF-Syntax würde den Rahmen sprengen und somit verweise ich auf eine gute Beschreibung dieser von Steven M. Rubin [22], im Anhang D seiner Publikation *"Computer Aids for VLSI Design*" [24, 23, 25].

```
 1 (edif AND_Example
 2  (edifVersion 2 0 0)
 3  (edifLevel 0)
 4  (keywordMap (keywordLevel 0))
 5  (status
 6   (written
 7    (timestamp 2003 06 16 14 21 37)
 8    (program "LeonardoSpectrum Level 3" (version "v2001_1d.45"))
 9    (author "Exemplar Logic Inc")))
10  (external PRIMITIVES
11   (edifLevel 0)
12   (technology (numberDefinition ))
13   (cell AND2 (cellType GENERIC)
14    (view INTERFACE  (viewType NETLIST)
15     (interface
16      (port p0 (direction INPUT))
17      (port p1 (direction INPUT))
18      (port out (direction OUTPUT))))))
19  (library work
20   (edifLevel 0)
21   (technology (numberDefinition ))
22   (cell AND (cellType GENERIC)
23    (view A2  (viewType NETLIST)
24     (interface
25      (port a (direction INPUT))
26      (port b (direction INPUT))
27      (port o (direction OUTPUT)))
28     (contents
29      (instance ix1 (viewRef INTERFACE  (cellRef AND2 (libraryRef PRIMITIVES ))))
30      (net a
31       (joined
32        (portRef a )
33        (portRef p0 (instanceRef ix1 ))))
34      (net b
35       (joined
36        (portRef b )
37        (portRef p1 (instanceRef ix1 ))))
38      (net o
39       (joined
40        (portRef o )
41      (portRef out (instanceRef ix1 ))))))))
42  (design AND_Example (cellRef AND (libraryRef work ))))
```

Abbildung 2.2: EDIF-Beispiel eines AND

2.2 Rekonfigurierbare Hardware

Wie schon in Abschnitt 1.1 beschrieben, gab es bis zum Erscheinen von konfigurierbarer Hardware eine grobe Unterteilung von Hardware in zwei Gebiete: ASIC und GPP. Beide Gebiete haben ihre Vor- und Nachteile und wer Hardware entwickelte, musste sich für eine dieser Lösungen entscheiden. Mit dem Erscheinen von konfigurierbarer Hardware Ende der 70er Jahre änderte sich das. Anfangs basierte die Konfigurierbarkeit dieser Chips noch auf kleinen Schmelzsicherungen, welche durch gezielte Überspannung zum Schmelzen gebracht wurden und auf diese Art die Schaltung realisierten. Dies war ein irreversibler Vorgang, doch immer noch preisgünstiger als die Herstellung fester Hardware.

Darauf folgten Chips, deren Programmierbarkeit auf EPROMs basiert. Ihr Vorteil liegt darin, dass ihre Programmierung mit Hilfe von UV-Licht gelöscht werden kann (100-200 Mal) und somit der Chip wiederverwendbar ist. Später entwickelte man dann rekonfigurierbare Hardware, welche ihre Konfigurationsdaten mit Hilfe von EEPROM oder Flash-Speicher speicherten. Da ihre Konfiguration nun nicht mehr auf dem physischen Zerstören von Verbindungen beruhte, sonder lediglich die alten Konfigurationsdaten gelöscht und neue geladen werden mussten, ergab sich daraus eine theoretisch unbegrenzte Wiederverwendbarkeit.

Im Laufe der Zeit erhöhte sich die Komplexität der rekonfigurierbaren Hardware immer mehr bis hin zu den heutigen FPGAs, welche neben normalen logischen Schaltelementen auch komplexere Komponenten wie z.B. Multiplizierer, RAM-Zellen oder sogar Mikroprozessoren enthalten (z.B. Xilinx Virtex2P.)
Fünf der wichtigsten Unterscheidungsmerkmale rekonfigurierbarer Hardware von heute sind: *Granularität*, *Tiefe der Programmierbarkeit*, *Rekonfigurierbarkeit*, seine *Schnittstelle* zur Außenwelt und das zugrundeliegende *Berechnungsmodel*.

- *Granularität*: Diese bezieht sich auf die Datenbreite, die von den rekonfigurierbaren Einheiten (RPU) bearbeitet werden kann und die damit verbundene Größe der einzelnen RPUs. Man unterscheidet hier hauptsächlich zwischen *fine-grain* (feinkörnigen) und *coarse-grain* (grobkörnigen) Architekturen. *Fine-grain* Architekturen bestehen normalerweise aus logischen Gattern, Flip-Flops und LUTs und arbeiten auf Bitebene, auf der sie boolsche Funktionen und Schaltwerke bis hin zu komplexen arithmetischen Operationen implementieren. Im Gegensatz dazu bestehen die RPUs einer *coarse-grain* Architektur aus kompletten Funktionseinheiten wie ALUs oder Multiplizierern und arbeiten mit einer wesentlich größeren Datenbreite (8, 16, 24, 32 Bit.) Systeme die beide Ansätze miteinander verbinden nennt man *mixed-grain* Architekturen.

- *Tiefe der Programmierbarkeit*: Sie gibt an, wie viele Konfigurationen in einer RPU gespeichert werden können, eine oder mehrere. Bei Systemen mit nur einer Konfiguration ist die Funktionalität einer RPU auf die momentan geladene Konfiguration beschränkt. RPUs mit mehrere Konfigurationen hingegen können mehrere Anwendungen ausführen, ohne das Konfigurationsdaten neu geladen werden müssen. Sie müssen lediglich die momentan aktive Konfiguration wechseln.

- *Rekonfigurierbarkeit*: Unter Rekonfiguration versteht man das erneute Laden einer Konfiguration. Dieser Vorgang ist entweder *statisch* (Ausführung der aktiven Konfiguration wird unterbrochen) oder *dynamisch* (läuft parallel zur momentan ausgeführten Konfiguration.) Typischerweise haben Systeme mit nur einer Konfiguration statische und dementsprechend Systeme mit mehreren Konfigurationen dynamische Rekonfigurierbarkeit. Dadurch sind dynamisch rekonfigurierbare RPUs in der Lage Konfigurationsdaten nachzuladen, während die momentan aktuelle Konfiguration noch ausgeführt wird, was die Zeitverzögerung bei einem Konfigurationswechsel erheblich verringert.

- *Schnittstelle*: Jedes rekonfigurierbare System benötigt eine Schnittstelle für die Ein- und Ausgabe von Daten und Konfigurationen. Zumeist ist ein solches System kein eigenständiges, sondern dient der Unterstützung eines Prozessors (Hostprozessor) in der Art eines Coprozessors. Ein rekonfigurierbares System hat eine *remote* Schnittstelle, wenn RPU und Hostprozessor nicht auf dem gleichen Chip sind. *Local* wird das Interface hingegen genannt, wenn RPU und Hostprozessor auf einem Chip sind oder die RPU in den Datenpfad des Hostprozessors eingebunden ist.

- *Berechnungsmodel*: Hierbei unterscheidet man zwischen dem klassischen 1-Prozessorsystem, SIMD, MIMD und VLIW.

Die heute am weitesten verbreiteten rekonfigurierbaren Chips sind FPGAs. Sie erlauben dem Entwickler die Manipulation einer Schaltung auf *gate*-Ebene wie z.B. Flip-Flops, LUTs oder anderen logischen Schaltelementen und machen FPGAs somit für die Realisierung komplexer bitorientierter Schaltungen lukrativ. Dennoch haben FPGAs auch ein paar Nachteile. Zum einen sind sie langsamer als ASICs und zum anderen bieten sie meistens keine effiziente Möglichkeiten für die Gestaltung grobgranularer Schaltungen, was sich momentan aber stark wandelt.

2.3 State of the art - vergleichbare Ansätze

Im Folgenden sollen neben der VHM auch einige andere Ansätze rekonfigurierbarer Hardware vorgestellt werden. Es handelt sich dabei um die Architekturen PACT XPP [3], Quicksilver [6] und MorphoSys [11]. Im Rahmen dieser Arbeit wurde keiner dieser drei Ansätze getestet, da mir keine der beschriebenen Plattformen zur Verfügung stand. Abgeschlossen wird dieses Kapitel mit einer Einführung in die VHM und die Funktionsweise des VHBC-Compilers, wie beides zu Beginn dieser Arbeit vorlagen, und einem Vergleich mit den vorher beschriebenen Architekturen.

2.3.1 PACT XPP

Die *eXtreme Processing Platform* (XPP) ist ein Produkt der PACT Informationstechnologie GmbH [3]. Abbildung 2.3 zeigt die allgemeine Struktur der XPP und soll einen groben Überblick über die Architektur verschaffen, welche im Weiteren detailierter erklärt werden soll.

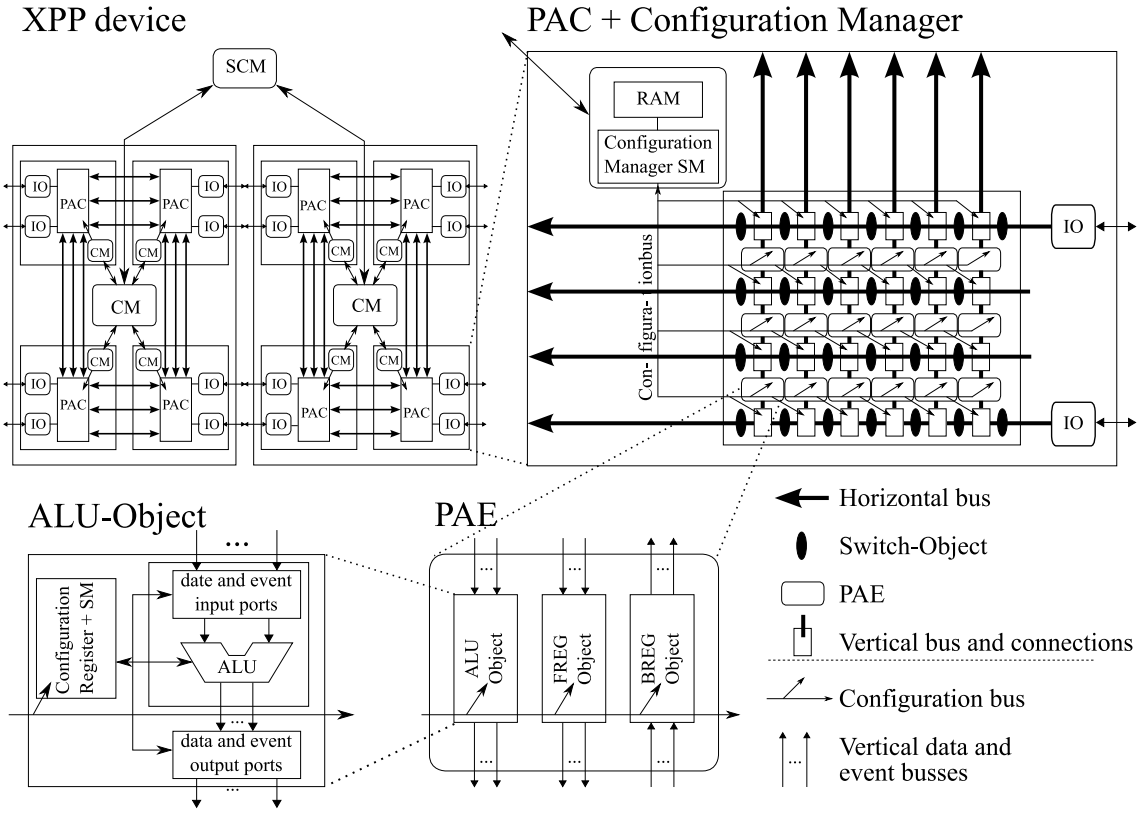


Abbildung 2.3: Struktur der XPP [4, 5]

Die Hauptkomponente der XPP besteht aus einem hierarchisch aufgebautem Feld von konfigurierbaren coarse-grain Elementen, die *Processing Array Elements* (PAEs, Abb. 2.3 unten rechts) genannt werden, und einem paketorientiertem Verbindungsnetzwerk. Die PAEs in der XPP sind in einem oder mehreren *Processing Array Clusters* (PACs, Abb. 2.3 oben rechts) zusammengefasst und jeder dieser PAC ist wiederum an einen *Configuration Manager* (CM) angeschlossen. Die CMs bilden untereinander eine Baumstruktur, an deren Spitze der *Supervising CM* (SCM)

steht und welche nicht an ein PAC angeschlossen ist. Ein CM besteht im wesentlichen aus einem Zustandsautomaten und internem RAM für Konfigurationsdaten. Die PAEs beinhalten die drei Komponenten: BREGs (*back*-Register), FREGs (*forward*-Register) und eine ALU (Abb. 2.3 unten links). Durch die Struktur des PACs ist es einem PAE möglich, mit allen anderen PAEs in der gleichen Zeile (horizontal) zu kommunizieren. Für die vertikale Kommunikation benutzt der XPP die B- und FREGs. Hierbei wird das BREG für die Kommunikation von unten nach oben und das FREG von oben nach unten verwendet. Die ALU führt Standard Festkommaberechnungen und Logikoperationen aus und implementiert weiterhin verschiedene 3-Input-Operationen wie Mult-Add (Multiplikation gefolgt von einer Addition, z.B. a*b+c), Sortieren und Zähler.

Im XPP-Prototypen XPU128-ES wurde weiterhin ein Speicherelement eingeführt, welches sowohl im FIFO-Modus betrieben als auch als normaler RAM benutzt werden kann (Abb. 2.4 rechts). Dieser Prototyp bestand aus zwei PACs mit je 64 ALU-PAEs (32 Bit), 16 Speicher-PAEs (je 1kB) und vier 32-Bit IO-Einheiten mit je zwei Kanälen. Zusätzlich wurde in jedes BREG eine ADD/SUB-Operation eingebaut. Abbildung 2 zeigt die allgemeine Struktur der XPP, der PACs mit ihrem CM, der PAEs und der ALUs.

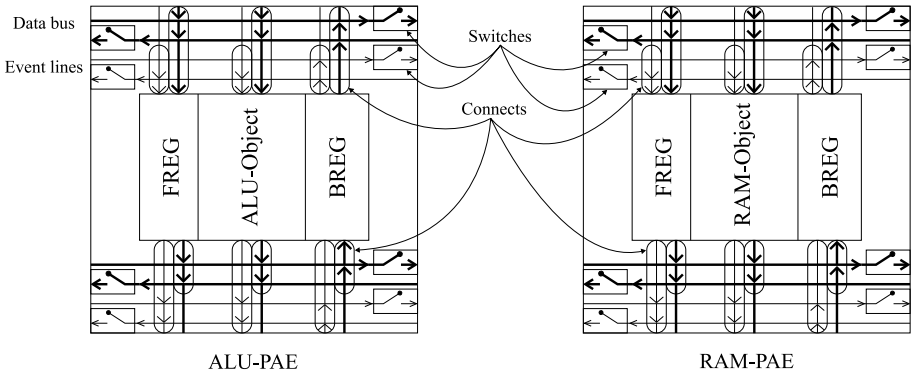


Abbildung 2.4: ALU-PAE und RAM-PAE, ihr Anschluss an die Daten- und Ereignisleitungen und Switches für die Unterbrechung der vertikalen Kommunikation [5]

Die Kommunikation zwischen den einzelnen Komponenten (PAEs, IO-Einheiten) erfolgt über ein paketorientiertes Netzwerk. Das Kommunikationsnetzwerk ist dabei so konstruiert, dass ein Paket pro Takt übertragen wird. Dabei wird durch das Netzwerk sichergestellt, dass keine Pakete verloren gehen. Auch dann nicht, wenn Leitungen belegt sind oder gerade eine Rekonfiguration durchgeführt wird. Bei den Paketen gibt es eine Unterscheidung zwischen Daten- und Ereignispaketen. Während Datenpakete eine der Architektur entsprechende Breite haben (z.B. 32 Bit), sind Ereignispakete in jeder Implementierung lediglich ein paar Bit breit. Das in einem Ereignispaket enthaltene Informations-Bit kann dazu verwendet werden, die Berechnung in den ALUs oder die Paketgenerierung zu steuern oder kann sogar eine Selbstkonfiguration der XPP initiieren. Weiterhin ist anzumerken, dass jede berechnende Einheit umgehend mit ihrer Berechnung beginnt, sobald alle ihrer Eingangsdaten vorhanden sind und das vorhergehende Ergebnisse weitergeleitet wurde. Dabei synchronisieren sich die PAEs selbständig.

Die Architektur wurde auf schnelle, häufige und für den Benutzer transparente Rekonfiguration

optimiert. Da jeder einzelne CM im CM-Baum eigenständig arbeitet, können die CM die Teile, für die sie zuständig sind, parallel und unabhängig voneinander konfigurieren. So ist es auch möglich eine PAE zu konfigurieren, während die anderen weiter ihre Berechnung durchführen. Die Konfigurationsdaten selbst durchlaufen den CM-Baum von der Wurzel, dem SCM, bis zum zuständigen CM. Jede PAE speichert ihren Konfigurationszustand lokal, wobei es zwei einnehmbare Zustände gibt, *configured* oder *free*. Einmal konfiguriert geht eine PAE in den Zustand *configured* und verhindert somit von dem CM erneut konfiguriert zu werden.

Unmittelbar nachdem sie den Zustand *configured* eingenommen hat, beginnt sie mit der Berechnung. Es ist somit auch möglich, dass eine erst teilweise konfigurierte Anwendung bereits mit der Berechnung anfängt, während der Rest der Anwendung noch konfiguriert wird. Das Kommunikationsnetzwerk übernimmt auch hier die Verantwortung dafür, dass keine Pakete verloren gehen.

Um eine PAE wieder in den Zustand *free* überführen zu können, wurde ein spezielles Ereignispaket, das '*reconfig*'-Ereignis, implementiert. Jede ALU besitzt einen Eingang für dieses Ereignis. Um nicht an jede einzelne PAE einer Anwendung, welche auf der XPP ausgeführt wird, ein Rekonfigurationsereignis senden zu müssen, wurde ein Verfahren implementiert, welches ein eingegangenes Rekonfigurationsereignis an alle angeschlossenen Komponenten weiterleitet. Dieses Rekonfigurationsereignis wird anschließend von allen Komponenten verarbeitet, sofern sie dafür konfiguriert wurden. Dieses Verfahren bewirkt, dass lediglich ein Rekonfigurationsereignis an eine PAE einer Anwendung geschickt werden muss, um alle an der Anwendung beteiligten PAEs in den Zustand *free* zu überführen.

Rekonfigurations- und Nachladeanfragen können von jeder CM ausgelöst werden. Auch die SCM ist in der Lage diese Aktionen auszulösen, wobei sie aber auch auf externe Signale zu reagieren vermag. Weiterhin ist eine Rekonfiguration durch ein entsprechendes Ereignispaket realisierbar, welches durch ein PAE generiert wurde. Dadurch ist eine laufende Anwendung in der Lage sich selbst umzukonfigurieren. So ist beispielsweise folgende Situation denkbar, in der eine Anwendung aus mehreren Phasen besteht und nach Ablauf einer jeden Phase diese eine Rekonfiguration auslöst, welche die nächste Phase lädt.

Um Anwendungen auf die XPP abbilden zu können, wurde eine eigene Sprache, die *Native Mapping Language* (NML), entwickelt. Mit ihrer Hilfe hat der Programmierer direkten Zugriff auf alle Eigenschaften, welche die Hardware der XPP bietet. In NML bestehen Konfigurationen aus Modulen, welche in einer strukturellen Hardwarebeschreibungssprache spezifiziert werden, ähnlich der strukturellen Beschreibung in VHDL. Desweiteren sind in einem NML-Programm Angaben für die Handhabung von Konfigurationen enthalten.

Ein komplettes NML-Programm besteht aus mehreren Teilen:

- einenm oder mehreren Modulen (unkonfiguriert) und einer Folge von vorkonfigurierten Modulen
- differenziellen Änderungen der Konfiguration
- Angaben, welche Ereignissignale auf Konfigurations- und 'prefetch'-Anfragen abbildet

Bei differenziellen Änderungen der Konfiguration handelt es sich um Abweichungen von der Ausgangskonfiguration. Dadurch lässt sich bei den Rekonfigurationsdaten Platz und bei der Rekonfiguration Zeit sparen. Neben der NML wurde auch noch ein C-Compiler entwickelt. Der C-Compiler für die XPP übersetzt C-Funktionen in NML-Module, wobei er aber auf eine Untermenge des Standard C Sprachumfangs eingeschränkt ist und eine XPP-spezifische IO-Bibliothek verwendet. Er ist somit in der Lage, mit Hilfe des NML-Mappers, C-Programme direkt auf die XPP abzubilden. Dabei benutzt er Vektorisierungstechniken, um handhabbare Programmteile zu bestimmen und sie in eine Art Datenflussdarstellung zu überführen, d.h. in Datenströme, welche (vom Speicher oder den IO-Ports) durch das Operatorennetz fließen. Der Compiler extrahiert hierfür die rechenintensiven Teile, fügt selbständig IO-Angaben hinzu und übersetzt diese Teile für die XPP. Der restliche Teil der Anwendung läuft auf dem Hostprozessor, der eine XPU[11] als eine Art Coprozessor nutzt. Die Kommunikation zwischen dem Hostprozessor und der XPU geschieht dabei über die IO-Ports der XPU, über Shared-Memory oder über eine externe Schnittstelle der SCM. Das Programm, welches vom Hostprozessor ausgeführt wird, lädt und ersetzt automatisch die Konfiguration, wodurch die physische Größe der XPU nicht die Größe der Anwendung beschränkt. Abbildung 2.5 zeigt den NML und den XPP C-Compiler Design Flow.

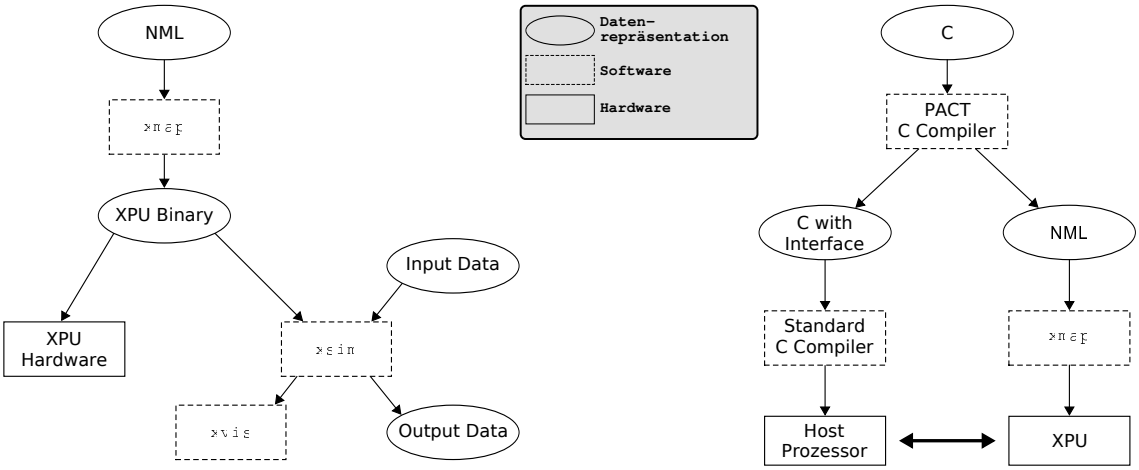


Abbildung 2.5: NML Design Flow (links) und XPP C Compiler Design Flow [4, 5]

Die erste kommerzielle XPU wird eine Weiterentwicklung des Prototypen XPU128-ES sein und mit einer Taktrate von 50 MHz laufen. Falls man alle Operationen nutzt (Mult-Add in ALU und

[11] E**x**treme **P**rocessing **U**nit: Bezeichnung für eine als Chip realisierte XPP

Add/Sub im BREG), so ist eine Zahl von 384 32 Bit Operationen pro Takt möglich, was zu einer Gesamtzahl von 19,2 GigaOps/Sek (bei 50 MHz) führen würde. Bei der Verwendung aller 16 IO-Kanäle ist eine theoretische Datenrate von 3 GB/Sek erreichbar. Dass es sich hierbei um theoretische Werte handelt, welche in der Realität wohl nicht erreicht werden, spiegelt sich auch in einigen Anwendungsimplementierungen wieder. So erreicht ein 128-Tap FIR-Filter mit 256 Operationen pro Takt eine Leistung von 12,8 GigaOps/Sek und der gleiche Filter mit komplexen Werten 14,7 GigaOps/Sek (290 Ops/Takt). Für die Zukunft haben sich die Entwickler der XPP vorgenommen, eine XPU mit 512 ALU-PAEs zu realisieren, welche auf 32 PACs aufgeteilt sind, einer CM-Baumtiefe von 4 oder 5 und einer Taktrate von 200 MHz. Sollten sie eine XPU mit diesen Eigenschaften bauen, so hätte diese eine theoretische Leistung von 300 GigaOps/Sek.

Nach den fünf, in Abschnitt 2.2 beschriebenen Unterscheidungsmerkmalen, ist die XPP-Architektur folgendermaßen einzuordnen:

- *Granularität=coarse grain*: Jede PAE entspricht einem ’coarse grain’-Element, welches Datenwörter mit einer Breite von bis zu 32 Bit verarbeitet.
- *Rekonfigurierbarkeit=statisch*: Es ist nicht möglich eine PAE zu konfigurieren, während sie zeitgleich Berechnungen durchführt. Dies wird durch die internen Zustände *configured* und *free* realisiert.
- *Tiefe der Programmierung=1*: Eine PAE enthält nur eine Konfiguration. Wurde sie konfiguriert, so nimmt sie den Zustand *configured* ein und verhindert damit ein weiteres Konfigurieren.
- *Schnittstelle*: Aus den Quellen ist nicht deutlich ersichtlich, ob es sich um eine *local*- oder eine *remote*-Schnittstelle handelt. Sehr wahrscheinlich ist aber eine *local*-Schnittstelle, da die XPP als Coprozessor fungieren soll und damit eine bessere Kommunikation zwischen Hostprozessor und XPP gewährleistet werden kann.
- *Berechnungsmodel=MIMD*: Allein die Tatsache, dass mehrere Anwendungen parallel auf der XPP ausgeführt werden können verdeutlicht, dass es sich um eine MIMD-Architektur handelt.

2.3.2 Quicksilver

Quicksilver Technologies nennt seine Architektur *Adaptive Computing Machine* (ACM). Die Hauptziele bei der Entwicklung der ACM waren:

- die Menge der Anwendungen für die ACM um solche zu erweitern, welche gewöhnlich mit Hilfe von ASICs realisiert werden
- die Leistung erheblich über die von DSP-Designs zu steigern
- Maximierung der Taktfrequenz und Minimierung des Energieverbrauchs
- Reduktion der Chipfläche
- alle zuvor genannten Ziele erreichen und trotzdem die Anpassungsfähigkeit für mehrere Echtzeitanwendungen erhalten

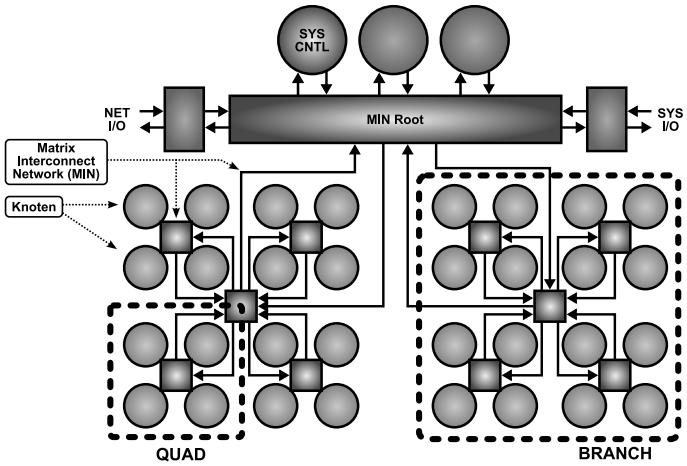


Abbildung 2.6: Beispiel der Struktur einer ACM [7]

Die zwei grundlegenden Komponenten der ACM sind zum einen die Knoten und zum anderen das *Matrix Interconnect Network* (MIN). Bei den Knoten handelt es sich um heterogene Knoten, die auf eine vorgegebene Klasse von Problemen optimiert wurden. Sie sind selbständig und mit eigenem Controller, Speicher und Berechnungsressourcen ausgestattet. Sowohl die Knoten, als auch das MIN arbeiten mit vollem Systemtakt (200-400 MHz bei einem von TSMC [8] im 13 μm-Verfahren hergestellten Chip). Die Konfiguration der Knoten geschieht mit Hilfe einer Binärdatei, welche *SilverWare* genannt wird. Sie unterscheidet sich im Hinblick auf die Größe von FPGA-Konfigurationsdaten erheblich, da sie aufgrund der Tatsache, dass nicht eine große Menge von *fine grain* Elementen sonder nur eine verhältnismäßig kleine Menge von *coarse grain* Elementen konfiguriert werden müssen, wesentlich kleiner ist und somit viel kürzere Rekonfigurationszeiten ermöglicht.

Alle der heterogenen Knoten haben den gleichen Grundaufbau. Sie bestehen aus dem *Node Wrapper*, einem Speicher und der *Algorithmic Engine.* Unterscheiden tun sie sich lediglich durch den

Aufbau ihrer *Algorithmic Engine*, den Aufbau des Speichers und der Verbindung zwischen den beiden. Abbildung 2.7 zeigt den allgemeinen Aufbau eines Knoten. Man sieht die Algorithmic Engine, den Speicher und den Node Wrapper (alles, was nicht Algorithmic Engine oder Speicher ist). Diese drei Komponenten sollen im Folgenden näher erklärt werden.

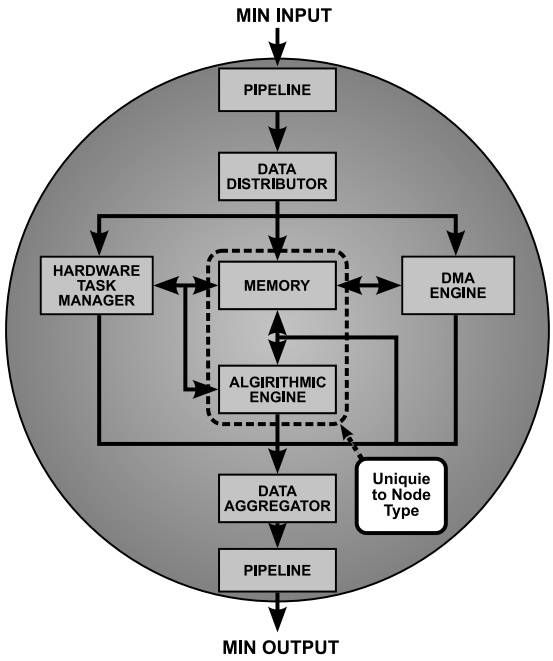


Abbildung 2.7: Allgemeiner Aufbau eines Knoten (Node Wrapper, Algorithmic Engine, Speicher) [7]

Jeder Knoten ist mit einem 16 KB großen Speicher ausgestattet, welcher zumeist in vier 1Kx32 Bitblöcken organisiert ist. Die Verbindung zwischen dem Speicher und der *Algorithmic Engine* ist nicht bei jedem Knotentyp gleich und auf die speziellen Bedürfnisse der *Algorithmic Engine* angepasst.

Der *Node Wrapper* besteht aus einer Schnittstelle zum MIN, einem Hardware Taskmanager und einer DMA-Engine. Er bietet für die heterogenen *Algorithmic Engines* eine immer gleiche Schnittstelle zum MIN und erweitert weiterhin alle Knoten um Dienste für das Taskmanagement und die Kommunikation mit anderen Knoten bzw. der Außenwelt. Durch diese, in allen Knotentypen enthaltenen Dienste, ist *SilverC*, die High-Level Programmiersprache für die ACM, in der Lage, von Systemfunktionen zu abstrahieren, mehrere Tasks pro Knoten zu managen, zu laden und zu entladen.

Die Hauptaufgaben des *Node Wrapper* sind das Taskmanagement und die Datenbehandlung. Als Task versteht man bei der ACM das kleinste Softwareelement, welches vom *Node Wrapper* behandelt werden kann. Er besteht aus einem Algorithmus oder einer Gruppe von Algorithmen, welche ohne Unterbrechung ausgeführt werden können, solange Daten vorhanden sind. Tasks können individuell geladen, entladen, gestartet oder gestoppt werden. Um mit Tasks umgehen zu können, enthält jeder Knoten einen Hardware-Taskmanager, der die Tasks in einer ’ready-to-run’-Queue verwaltet. Auch wenn eine *Algorithmic Engine* nur einen Task zur selben Zeit bearbeiten kann,

so ist ein Knoten in der Lage an bis zu 31 Tasks mitzuwirken. Auch hierbei übernimmt der *Node Wrapper* die kritische Aufgabe Daten für die Tasks (aktiv oder inaktiv) zu akzeptieren und an die richtige Stelle im Speicher zu schreiben. Der *Node Wrapper* ist in der Lage, bis zu 32 Ein- und 32 Ausgänge zu behandeln und Daten zu allen anderen Knoten zu senden bzw. von allen anderen Knoten zu empfangen, auch zu/von sich selbst. Dabei können diese Ein- und Ausgänge jedem der 31 Tasks zugewiesen werden. Zusätzlich sind auch noch für jeden Ein- und Ausgang Zähler und *Data Address Generators* (DAGs) vorhanden. Wenn nun Daten durch das MIN an den Eingängen eines Knoten ankommen, so wird der mit diesem Eingang verbundene DAG verwendet, um die Daten an die richtige Stelle des Speichers zu schreiben. Gleichzeitig werden die Daten des DAG für die nächsten Eingangsdaten aktualisiert. Für jeden Task eines Knotens hält der Hardware-Taskmanager (HTM) einen der vier möglichen Zustände: *suspend*, *idle*, *ready* oder *run*. Hierbei implementiert der HTM ein ’first-come, first-serve’, nonpreemptive Multitaskingsystem und erlaubt asynchrones Laden von Eingabedaten für Tasks im Zustand *idle*, *ready* und *run*. Weiterhin ist im *Node Wrapper* eine DMA-Engine enthalten. Sie erlaubt dem Systemcontroller den Node Wrapper anzuweisen, Programm- und Transferdaten in und aus dem Speicher zu übertragen, ohne die Algorithmic Engine oder den Systemcontroller zu verwenden oder zu unterbrechen.

Die *Algorithmic Engine* ist die Komponente eines Knoten, welche ihn hauptsächlich von den anderen Knotentypen unterscheidet. Sie ist an ihr späteres Anwendungsgebiet angepasst und auf dieses optimiert und somit auch fast nur für dieses zu verwenden. Grob unterteilt gibt es drei Hauptklassen von Knoten: ’*Adaptive*’ Knoten, ’*Domain*’ Knoten und ’*Programmable*’ Knoten. Die ’*Adaptive*’ Knoten sind zur Unterstützung von aufwendigen algorithmischen Berechnungen, welche komplexer Kontrolle bedürfen und mit Hilfe eines ’*Domain*’ Knoten nur wesentlich ineffizienter realisiert werden könnten, was sich in einem höheren Verbrauch an Fläche und Energie widerspiegelt. ’*Domain*’ Knoten sind für die Behandlung komplexer Algorithmen oder zumindest von Teilen komplexer Algorithmen konzipiert, z.B. Verschlüsselungsalgorithmen. Sie sind bzgl. Geschwindigkeit, Fläche und Energieverbrauch vergleichbar mit ASICs und ebenfalls wie diese nur für eine begrenzte Menge von Algorithmen verwendbar. Kontrolliert werden sie durch einen Zustandsautomaten. Um größere Code-Teile abzuarbeiten, welche nicht viel Rechenleistung benötigen, wurden die ’Programmable’ Knoten entworfen. Zu den typischen Anwendungen, die durch sie ausgeführt werden, gehören Betriebssysteme, GUI-Support und die Bearbeitung höherer Protokollebenen. Da es sich bei diesen Knotentypen nur um eine Art Container handelt, ist eine Vielzahl verschiedener Knoten möglich. Prinzipiell steht es jedem offen, seine eigene *Algorithmic Engine* zu entwickeln. So zeigt die folgende Tabelle die implementierten Knoten in der Adapt2400 ACM.

Beim *Matrix Interconnect Network*, kurz MIN, handelt es sich um ein homogenes Netzwerk, welches die Knoten miteinander verbindet und die Übertragung von Daten, *SilverWare* und Kontrollinformationen zwischen ihnen ermöglicht. Jede Verbindung zum MIN, sei es von den Knoten oder einer anderen Schnittstelle, besteht aus zwei unidirektionalen 32 Bit Datenleitungen, eine für eingehende und eine für ausgehende Daten. Zu jedem übertragenen 32 Bit Datenwort gibt es Adress- und Serviceinformationen, welche über eine separate Signalleitung übertragen werden. Diese In-

Tabelle 2.1: in der ACM Adapt2400 implementierte Knoten [7]

Knotentyp		Leistungsmerkmale
AXN	Adaptable Execution Node	◦ MAC und ALU Operationen mit variabler Wortbreite - acht 16-Bit Datenpfade (128 Bit insgesamt) - 16 8-Bit programmierbare Array-Multiplizierer - vier 16-Bit ALUs - acht 40-Bit Einheiten die als Accumulatoren, Shifter oder Zero/SaturateRound (ZSR) fungieren - acht Register Files mit vier 16-Bit Mehrzweckregistern - vier Data Address Generators die vier simultane Datenströme ermöglichen
DBN	Domain Bit Manipulation Node	- Bitmanipulation und wortorientierte 8-Bit Operationen - hohe Leistung bei komplexen Operationen, z.B. kryptographische Algorithmen wie SHA-1, 3DES, AES
PSN	Programmable Scalar Node	- Standard 32-Bit RISC-Architektur - 32-Bit load/store Adressbus - vierstufige Pipeline, bedingte und unbedingte Sprünge - maskierbare und unmaskierbare Interrupts - Little-Endian Byteadressierung - 1kB Instruktionen- und Datencache mit einer Zeilenbreite von 64 Byte
XMC	External Memory Controller	- kann bis zu 64 Tasks bedienen (normal 31) - 64 Ein- und 64 Ausgänge (normal je 32) - unterstützt bis zu 256 MB DDR SDRAM, 8 MB Flash und 2 MB statisches RAM - Point-to-Point Quelle und Ziel - Memory Random Access (MRA) lesen und schreiben - DMA

formationen werden vom MIN benutzt, um das Datenwort durch das Netzwerk zum richtigen Empfänger zu leiten. Die Grundkomponente des MIN ist ein 5-Port Netzwerk (siehe Abbildung 2.8). Dieses wird, falls mit 4 Knoten verbunden, Quad genannt. Falls es an einem der Ports zu einer Konkurrenzsituation zwischen zwei oder mehr Knoten kommen sollte, so wird der Eingang nach dem Round-Robin-Verfahren bestimmt. Ein Datentransfer innerhalb eines Quad dauert lediglich einen Takt. Das Zentrum des Matrix Interconnect Network bildet die MIN-Root, welche sich leicht von den anderen 5-Port Netzwerken unterscheidet. So besitzt sie 7 Ports und verfügt weiterhin über einen NetIO-Port, der ihr Zugriff auf den sogenannten Real-Time Eingang bietet. Weiterhin ist der NetIO-Port in der Lage, mittels Broadcast an alle oder eine ausgewählte Gruppe von Knoten Daten zu senden und stellt im MIN den Dienst mit der höchsten Priorität dar. Die MIN-Root kann ebenfalls noch an Speichercontroller oder andere externe Schnittstellen angeschlossen werden, wobei ein Port immer für den Systemcontroller reserviert ist. Die 5-Port Netzwerke können kaskadiert werden, d.h. es können bis zu vier 5-Port Netzwerke an die MIN-Root angeschlossen werden und an jedes von diesen wieder vier, bis an die äußersten 5-Port Netzwerke Knoten angehangen werden. Dieser fraktale Aufbau ermöglicht ein einfaches Skalieren der ACM.

Nach den fünf, in Abschnitt 2.2 beschriebenen Unterscheidungsmerkmalen, ist die QuickSilver-Architektur folgendermaßen einzuordnen:

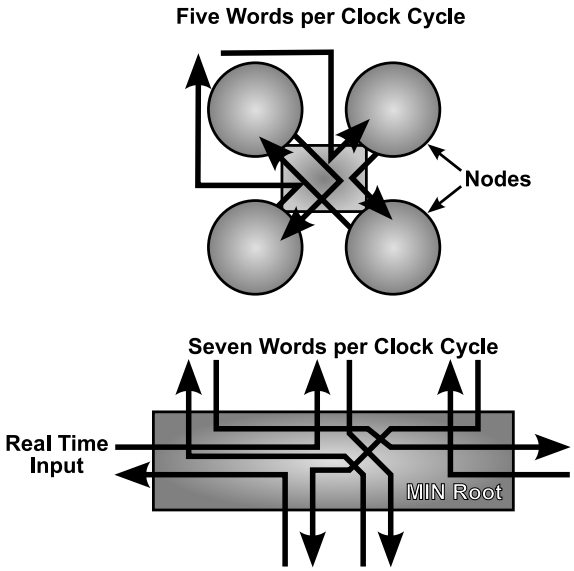


Abbildung 2.8: MIN Grundkomponenten: 5-Port Netzwerk (oben, bildet hier einen Quad) und die MIN-Root [7]

- *Granularität=coarse grain*: Jeder Knoten entspricht einem ’coarse grain’-Element, welches Datenwörter mit einer Breite von bis zu 32 Bit verarbeitet.
- *Rekonfigurierbarkeit=dynamisch*: Es ist möglich, Tasks auszutauschen (zu rekonfigurieren), während ein anderer Task ausgeführt wird.
- *Tiefe der Programmierung=31*: Jeder Knoten ist in der Lage, an bis zu 31 Tasks beteiligt zu sein.
- *Schnittstelle*: Die QuickSilver-Architektur benötigt keinen unterstützenden Prozessor, da sie in der Lage ist alle Aufgaben (Rekonfiguration, Laden von Daten, Ausgabe von Ergebnissen, etc.) selbst zu erledigen.
- *Berechnungsmodel=MIMD*: Die QuickSilver-Architektur entspricht am ehesten dem MIMD-Model, da jeder Knoten eine andere Operation auf verschiedenen Daten ausführen kann.

2.3.3 MorphoSys

MorphoSys ist eine rekonfigurierbare Architektur für datenparallele berechnungsintensive Anwendungen. Sie wurde an einer Abteilung der University of California, Irvine [9], der Henry Samueli School of Engineering [10], entwickelt und hatte zum Ziel, die Flexibilität dieses Modells zu studieren.

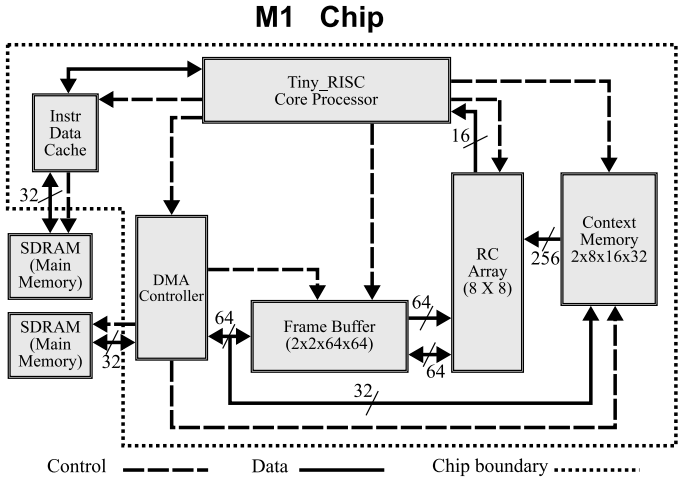


Abbildung 2.9: Blockdiagramm der MorphoSys-Implementierung M1 [12]

Das MorphoSys-System besteht aus einem Feld von rekonfigurierbaren Zellen (RC-Array) mit Konfigurationsspeicher (Context Memory), einem Kontrollprozessor (TinyRISC), Datenpuffern (Framebuffer) und einem DMA-Kontroller. Die Architektur ist so konzipiert, dass der TiniyRISC die sequenziellen Teile einer Anwendung ausführt, während das RC-Array für die parallelisierbaren und berechnungsintensiven Teile verantwortlich ist.

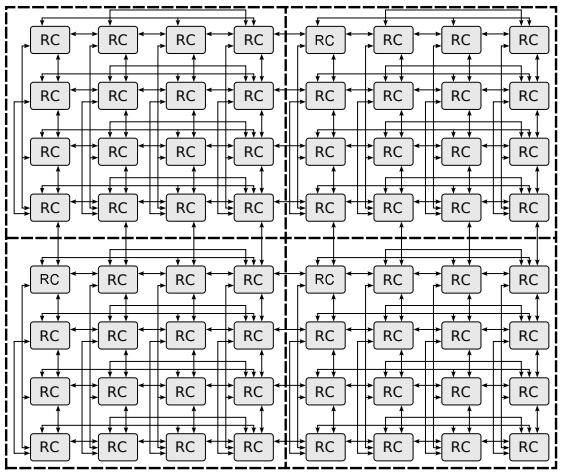


Abbildung 2.10: 8x8 RC-Array mit 2D-Mesh und kompletter Verbindung der Quadranten [12]

Das RC-Array bildet die Hauptkomponente des MorphoSys. Es besteht aus einem Feld von 8x8 konfigurierbaren coarse-grain Zellen, welche in einer SIMD-Weise angeordnet sind. Die Steuerung des RC-Arrays erfolgt mit Hilfe des TinyRISC-Prozessors. Im Gegensatz zu XPP und Quick-

silver sind beim MorphoSys alle RCs identisch und bestehen aus einem ALU-Multiplizierer, einem Shifter, zwei Input-Multiplexern, einem Kontextregister (32 Bit), einem Ausgaberegister (32 Bit), einem Feedback-Register und einem Registerfile mit vier 16-Bit Registern.

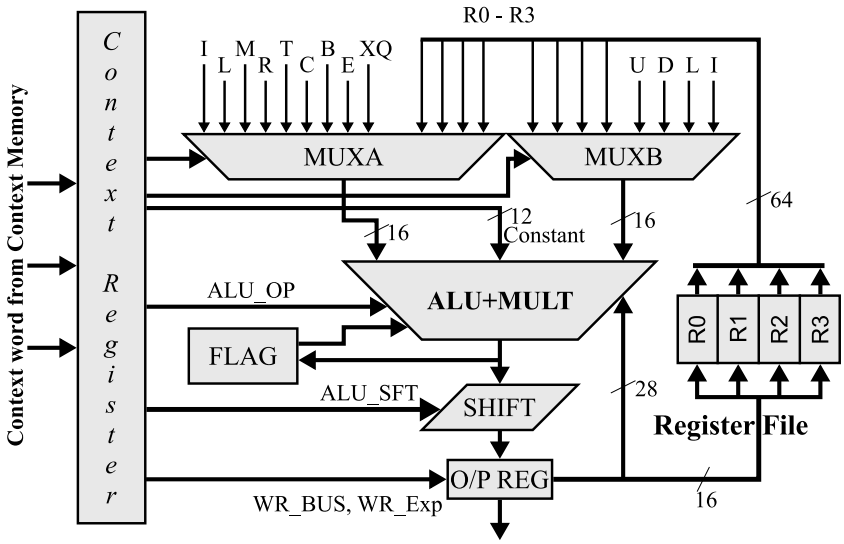


Abbildung 2.11: Aufbau einer RC [12]

Der ALU-Multiplizierer ist zusammengesetzt aus einem 16x12 Multiplizierer und einer 16-Bit ALU, wobei der Addierer der ALU mit 28-Bit Eingängen ausgestattet ist, um einen Verlust an Genauigkeit bei MAC-Operationen zu verhindern, da die Ausgabe des Multiplizierers bis zu 28 Bit groß sein kann. Neben den arithmetischen und logischen Standardoperationen verfügt die ALU auch noch über Funktionen wie die Berechnung des Absolutwertes der Differenz zweier Operanden oder eine 1-Takt MAC-Operation. Insgesamt sind in der ALU einer RC 25 Funktionen implementiert. Die Input-Multiplexer dienen der Auswahl der richtigen Eingabedaten und werden mittels Kontrollbits aus dem Kontextwort gesteuert. Zur Auswahl stehen dabei folgende Eingänge:

- die Ausgänge der vier benachbarten RCs (nearest neighbor)
- die Ausgänge der RCs in der gleichen Zeile/Spalte aus dem gleichen Quadranten
- der '*expresslane*' Datenbus (Abb. 2.13)
- die Register der Registerfiles der RC

Die alles steuernde Komponente des MorphoSys ist ein 32-Bit Prozessor, der als TinyRISC bezeichnet wird. Dieser bearbeitet die allgemeinen, sequenziellen und nicht für das RC-Array geeigneten Aufgaben, trägt die Verantwortung für alle Datentransfers zu und vom Framebuffer, sowie für das Laden von Konfigurationen, und steuert das RC-Array. Um diese Aufgaben effektiv erledigen zu können, wurde der TinyRISC um einige Instruktionen erweitert. Es handelt sich hierbei um Instruktionen für den Datentransfer zwischen Hauptspeicher und Framebuffer, zum Laden von Kontextworten vom Hauptspeicher in den Kontextspeicher und um Instruktionen für die Kontrolle der Ausführung der RC-Arrays.

Eine weitere wichtige Komponente der MorphoSys-Architektur ist der Framebuffer. Er fungiert als interner Datenspeicher, welcher den Speicherzugriff für die RCs transparent gestaltet. Der Framebuffer selbst ist in zwei logische Blöcke unterteilt, um Berechnungen und Datentransfers gleichzeitig ausführen zu können. Genauer gesagt bietet einer der beiden Blöcke Berechnungsdaten, während der andere Block Daten in den externen Speicher überträgt und neue Daten für die nächste Berechnung holt. Jeder Block kann 128 Zeichen mit je 8 Byte speichern, was zu einer Gesamtgröße des Framebuffers von 2 kByte führt.

Der Kontextspeicher beinhaltet die Kontexworte für das RC-Array für jeden Takt und ist ebenfalls in zwei Blöcke unterteilt, je einen für Zeilen- und einen für die Spaltenkontextworte. Jeder Block besteht aus 8 Teilen, wobei ein Teil zu einer Zeile/Spalte im RC-Array gehört. Jedes dieser 8 Teile kann wiederum 16 Kontextworte speichern. Eine Konfiguration der RC-Arrays wird auch Konfigurationsebene genannt und besteht aus 8 Kontextworten aus entweder dem Zeilen- oder dem Spaltenblock, was eine Maximalzahl von 16 Konfigurationsebenen pro Block und 32 Ebenen insgesamt bedeutet. Eine weitere interessante Eigenschaft ist der sogenannte Kontext-Broadcast. Durch ihn wird an alle RCs in einer Zeile bzw. Spalte das gleiche Kontextwort gesendet und bewirkt, dass alle RCs in dieser Zeile/Spalte die gleiche Operation ausführen, nur auf verschiedenen Daten (SIMD). Die Kontextworte selbst konfigurieren die RCs und sind die Grundlage für die Programmierung des Verbindungsnetzwerkes. Sie werden im Kontextregister abgelegt, welches im Gegensatz zum Kontextspeicher in jeder RC enthalten ist.

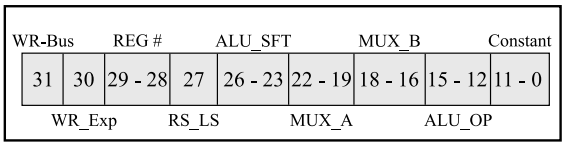


Abbildung 2.12: Aufbau der Kontextwortes einer RC [12]

Das Kontextwort (Abb. 2.12) ist ein 32 Bit Wert mit allen, für den Ablauf notwendigen Daten.

- WR_Bus: gibt an, ob die Ausgabe über den Datenbus in den Frambuffer geschrieben wird
- WR_EXP: gibt an, ob die RC auf Zeilen/Spalten '*express lane*' schreibt
- ALU_SFT: gibt an, um wieviel Bit das ALU-Ergebnis verschoben wird
- RS_LS: gibt an, in welche Richtung das ALU-Ergebnis verschoben wird
- REG#: gibt das Register aus dem Registerfile an, in welches das Ergebnis geschrieben wird
- MUX_A/MUX_B: Kontrollbits für die Multiplexer an den Eingängen
- ALU_Op: gibt die Funktion der ALU-Einheit an
- Constants: Platz für Konstanten (z.B. für Multiplikationen mit einer Konstanten, ...)

Das Verbindungsnetzwerk (Abb. 2.10) des MorphoSys teilt sich in 3 hierarchische Ebenen auf: das RC-Array-Netzwerk, das Intra-Quadrant-Netzwerk und das Inter-Quadrant-Netzwerk. Beim RC-Array-Netzwerk handelt es sich um das grundlegende Netzwerk, welches eine 'nearest neighbor' Vernetzung zwischen den einzelnen RCs realisiert. Unter einem Quadranten versteht man beim MorphoSys eine Feld von 4x4 RCs und innerhalb dieser Quadranten kann jede RC auf die Ergebnisse einer jeden anderen RC in der gleichen Zeile/Spalte mit Hilfe des Intra-Quadrant-Netzwerkes zugreifen. Das Inter-Quadrant-Netzwerk erlaubt die Kommunikation zwischen aneinandergrenzenden Quadranten über die '*express lane*' (Abb. 2.13). Es verläuft zeilen- und spaltenweise und stellt Ergebnisse von einer von vier RCs einer Zeile/Spalte eines Quadranten allen vier RCs in der gleichen Zeile/Spalte eines angrenzenden Quadranten zur Verfügung.

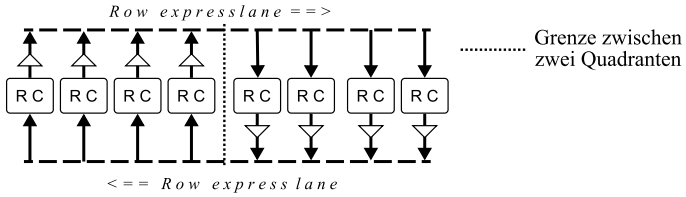


Abbildung 2.13: Funktionsweise der '*express lane*' [12]

Eine grundlegende Eigenschaft der MorphoSys-Architektur ist, dass alle RCs in einer Zeile (oder Spalte) das gleiche Kontextwort verwenden, wohl aber auf verschiedenen Daten arbeiten.

Nach den fünf, in Abschnitt 2.2 beschriebenen Unterscheidungsmerkmalen, ist die MorphoSys-Architektur folgendermaßen einzuordnen:

- *Granularität=coarse grain*: MorphoSys wurde entworfen, um 8 oder 16-Bit Daten zu verarbeiten. Um eine effektive Verarbeitung zu gewährleisten, entschied man sich zwangsläufig für eine coarse-grain Architektur.
- *Rekonfigurierbarkeit=dynamisch*: Dem MorphoSys ist es möglich, während das RC-Array arbeitet, Daten in einen nicht aktiven Teil des Kontextspeichers zu laden, ohne das RC-Array unterbrechen zu müssen. Dabei ist es der TiniyRISC, der die Lade- und Nachladevorgänge für die Kontextworte einleitet und der DMA-Controller, der sie ausführt.
- *Tiefe der Programmierung=32*: Der Kontextspeicher bietet Platz für bis zu 32 Konfigurationsebenen, wobei es dem Benutzer obliegt, die Konfigurationen über die Zeilen oder die Spalten zu verteilen.
- *Schnittstelle=local*: Der TinyRISC befindet sich auf dem gleichen Chip wie das RC-Array.
- *Berechnungsmodel=SIMD*: Das RC-Array ist für SIMD-Operationen ausgelegt. Dennoch ist es möglich, das RC-Array unkonfiguriert zu lassen und nur den TinyRISC für Berechnungen zu verwenden, was aber nicht im Sinne der MorphoSys-Architektur wäre.

2.4 VHM und VHBC - Vorausgegangene Arbeiten

Bei der *Virual Hardware Machine* (VHM) handelt es sich um einen Hardwareprozessor, welcher in der Lage ist, eine durch den *Virtual Hardware Byte Code* (VHBC) repräsentierte abstrakte Hardwarebeschreibung auszuführen. Der VHBC und das Konzept für eine VHM war das Ergebnis der Diplomarbeit von Sebastian Lange [1]. Seine Aufgabe war es, ein funktionierendes und schlüssiges Konzept für eine Virtuelle Hardware Machine (VHM) vorzulegen und einen Compiler zu entwerfen, welcher VHDL in VHBC übersetzt. Um nicht den Rahmen seiner Diplomarbeit zu sprengen, wurde der Compiler so gestaltet, dass er nur VHDL, basierend auf der SimPrim-Bibliothek verarbeitete. Da diese aus einer festgelegten Menge von Komponenten besteht, deren Namen den genauen Typ und die Größe der Komponente angibt, vereinfacht das das Parsen und Weiterverarbeiten enorm.

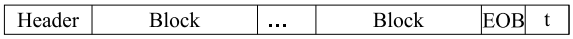


Abbildung 2.14: Allgemeiner Aufbau des VHBC [1]

Der VHBC (Abb. 2.14) besteht aus einem Header, gefolgt von einer oder mehreren Instruktionsebenen. Der Header (Abb. 2.15) beinhaltet Informationen über die ihm folgenden Instruktionsebenen und Informationen, welche dabei helfen, die Instruktionsebenen zu verarbeiten. Dabei handelt es sich im einzelnen um Informationen wie Versionsnummer (h_vers, l_vers), Adressbreite (addr_width), Anzahl der verwendeten Register (reg_number), Anzahl der Ein- und Ausgabeports (in-/out_port_num) sowie Informationen über die verwendeten Konstanten. Das 'VHBC' am Anfang des Headers dient lediglich der Identifikation einer VHBC-Datei als solche.

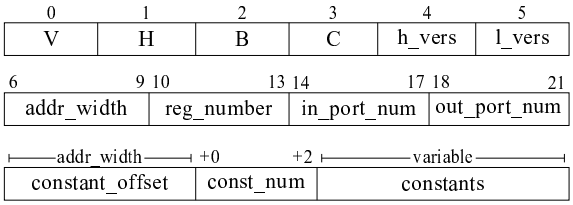


Abbildung 2.15: Aufbau des VHBC-Headers [1].

Die auf den Header folgenden Instruktionsebenen (Abb. 2.16) beschreiben die eigentliche virtuelle Hardware. Eine Instruktionsebene besteht dabei aus voneinander unabhängigen Instruktionen, welche parallel ausgeführt werden können. Jede Instruktion (Abb. 2.17) repräsentiert dabei ein Hardwareelement, welches durch einen Operationscode und zwei Operandenadressen näher spezifiziert wird. Abgeschlossene wird eine Instruktionsebene mit einer '*End of Block*'-Instruktion (EOB), wobei zwei EOB-Instruktionen hintereinander das Ende des VHBC markieren. Die Anzahl der Instruktionsebenen und deren Länge sind nicht festgelegt.

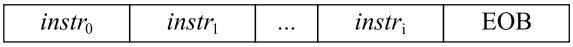


Abbildung 2.16: Aufbau einer Instruktionsebene [1].

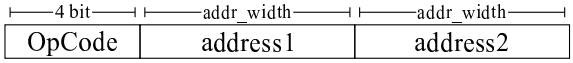


Abbildung 2.17: Aufbau einer Instruktion [1].

Tabelle 2.3: Die im VHBC implementierten Operationscodes [1]

OpCode		Mnemonic	mathematischer Ausdruck (a,b=Inputs;c=Output)
0_{10}	0000_2	EOB	N/A
1_{10}	0001_2	NOP	$c = c$
2_{10}	0010_2	MOV	$c = a$
3_{10}	0011_2	NOT	$c = \overline{a}$
4_{10}	0100_2	AND	$c = a \wedge b$
5_{10}	0101_2	NAND	$c = \overline{a \wedge b}$
6_{10}	0110_2	OR	$c = a \vee b$
7_{10}	0111_2	NOR	$c = \overline{a \vee b}$
8_{10}	1000_2	XOR	$c = (\overline{a} \wedge b) \vee (a \wedge \overline{b})$
9_{10}	1001_2	EQ	$c = (a \wedge b) \vee (\overline{a} \wedge \overline{b})$
10_{10}	1010_2	IMP	$c = (a \wedge b) \vee \overline{a}$
11_{10}	1011_2	NIMP	$c = a \wedge \overline{b}$
12_{10}	1100_2	CMOVE	$c = \begin{cases} b \text{ if } a \text{ true} \\ c \text{ otherwise} \end{cases}$

Mit der Spezifikation des VHBC wurde auch die Architektur der VHM festgelegt. So handelt es sich bei der VHM nicht wie bei den meisten anderen Ansätzen (vgl. Abschnitte 2.3.1, 2.3.2, 2.3.3) um eine Datenflussarchitektur, in der mehrere konfigurierbare Elemente in einer Netz- oder Baumstruktur miteinander verbunden sind, sondern eher um eine zeilenähnliche Anordnung der Funktionseinheiten (*Function Unit*=FU), welche ihre Daten aus einem Registerfile beziehen, anschließend parallel ihre Berechnungen auf diesen Daten ausführen und ihr Ergebnis wieder in das Registerfile übertragen (verdeutlicht in Abb. 2.18). Dabei kann jede FU auf alle Register des Registerfiles lesend zugreifen. Schreiben kann sie aber lediglich in das, ihrer Position entsprechende Register, d.h. FU *i* schreibt sein Ergebnis in Register *i*.

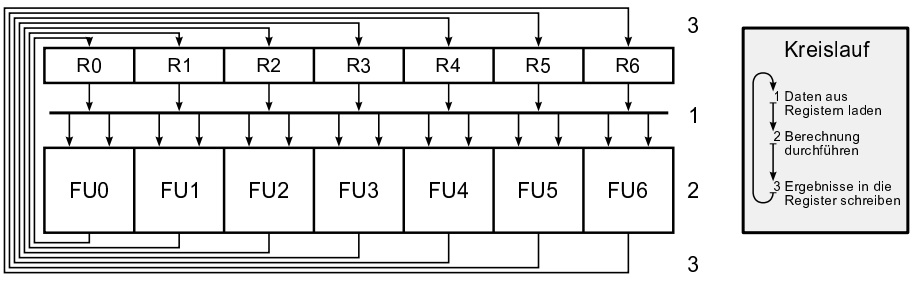


Abbildung 2.18: Ablauf der Berechnung innerhalb der VHM

Die erste Implementierung der VHM in Hardware schuf Nick Bierwisch in seiner Diplomarbeit [2]. Er erarbeitete, auf der Grundlage der Ideen und Vorschläge von S. Lange, eine VHDL-Beschreibung der VHM, welche er aber leider nicht auf einem FPGA implementieren konnte. Dennoch sind die Erkenntnisse aus seiner Arbeit sehr wichtig, da sie die Engpässe der VHM aufdecken und dieses späteren Entwicklern die Möglichkeit gibt, diese Engpässe gezielt zu umgehen bzw. zu beseitigen. Es handelt sich dabei z.B. um die Tatsache, dass jede FU lesend auf alle Register zugreifen kann, was in der Hardwareimplementierung zu einer hohen Anzahl von großen Multiplexern führt. So würde eine VHM mit 64 FUs und 64 Registern insgesamt 2*64 64-1 Multiplexer benötigen, was einen enormen Hardwareaufwand bedeutet. Ein weiterer Kritikpunkt ist der feingranulare Charakter der VHM, da man mit der Implementierung von ausschließlich logischen Operationen mit der VHM lediglich eine große und komplexe Maschine für die Realisierung von logischen Funktionen besitzt.

Der Aufbau der ersten VHM entsprach im Wesentlichen den Vorgaben von S. Lange. Der Arbeitsablauf dieser ersten VHM lässt sich grob in zwei Phasen unterteilen. In der ersten Phase liest die VHM den VHBC ein und initialisiert die Maschine. Dieser Vorgang muss nur ein einziges Mal ausgeführt werden und ist erst wieder notwendig, wenn ein neuer VHBC geladen werden soll. Dieser Dekodiervorgang stellte in Nick Bierwischs VHM den komplexesten Teil dar. Die zweite Phase ist eine Endlosschleife, in der die eigentliche Berechnung durch die im VHBC-Image beschriebe Schaltung geschieht. Diese Phase wird auch Makrozyklus genannt und besteht aus dem Einlesen der Eingabewerte, einer, durch die Menge der Instruktionsebenen im VHBC festgelegten Anzahl von Mikrozyklen und dem Schreiben der Ausgabewerte. Ein Mikrozyklus entspricht dabei genau der Abarbeitung einer Instruktionsebene. Abbildung 2.19 verdeutlicht den Arbeitsablauf noch einmal grafisch. Für eine detaillierte Beschreibung sei auf die beiden Diplomarbeiten verwiesen [1, 2].

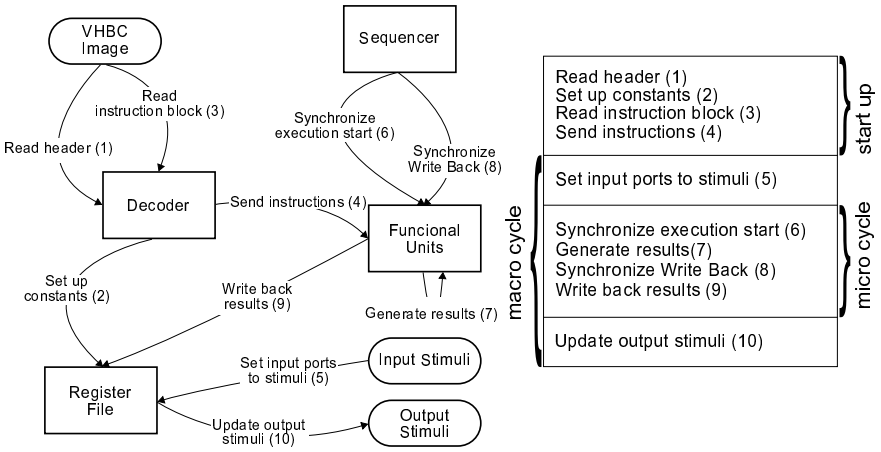


Abbildung 2.19: Schritte bei der Ausführung eines VHBC-Images [1]

Nach den fünf, in Abschnitt 2.2 beschriebenen Unterscheidungsmerkmalen, ist die VHM von S. Lange folgendermaßen einzuordnen:

- *Granularität=fine grain*: Jede FU ist lediglich in der Lage logische Operationen auf maximal zwei 1-Bit Werte anzuwenden.
- *Rekonfigurierbarkeit*=statisch: Jede Rekonfiguration unterbricht die VHM in ihrer Arbeit. Die Abarbeitung wird erst wieder aufgenommen, wenn die Konfiguration abgeschlossen ist.
- *Tiefe der Programmierung*=1: Auch wenn jede FU einen Instruktionsspeicher für 16 Instruktionen besitzt, so gibt es doch nur eine, in der VHM gehaltene Konfiguration.
- *Schnittstelle*=?: Es gibt keine (hardwareseitige) Implementierung der VHM.
- *Berechnungsmodel=MIMD*: Die Aufbau der VHM und des VHBC sind für MIMD-Berechnungen ausgelegt.

Diese fünf Eigenschaften zeigen, dass die VHM in vielen Punkten noch stark verbesserungswürdig ist.

Kapitel 3

Ansatz

Wie der Vergleich mit anderen Realisierungen rekonfigurierbarer Hardware aus Kapitel 2 zeigt, gibt es viele verbesserungsfähige Eigenschaften der VHM. Um deren Leistungsfähigkeit zu erreichen ist es unabdinglich, die Schwachpunkte der VHM zu lokalisieren und zu entfernen, bzw. die VHM im Rahmen der Möglichkeiten dieser Arbeit zu verbessern. Dieses Kapitel beschäftigt sich mit den Schwachpunkten der ersten Implementierungen der VHM und des VHBC-Compilers und zeigt entsprechende Lösungsmöglichkeiten, welche in die neue VHM einfließen werden.

3.1 Die Schwachpunkte der alten VHM

Der Vergleich der in dieser Arbeit verwendeten fünf Unterscheidungsmerkmale für rekonfigurierbare Hardware (Granularität, Rekonfigurierbarkeit, Tiefe der Programmierung, Schnittstelle, Berechnungsmodel; siehe Kapitel 2) zeigt gleich mehrere Schwachpunkte. Am meisten fällt die *fine grain* Architektur auf. Auch wenn es theoretisch möglich ist jede beliebige Berechnung mit Hilfe von Logikoperationen zu realisieren, so hat dies praktisch keinen Sinn, da sowohl die Konfigurationsdaten als auch die VHM, selbst für relativ einfache Berechnungen, viel zu große Ausmaße annehmen würden. Das folgende Beispiel soll dies verdeutlichen. Die momentane Realisierung der VHM besitzt mit 32 Funktionseinheiten und einem Instruktionenspeicher von 16 Instruktionen pro Funktionseinheit, nur ausreichend Ressourcen für zwei 6 Bit Volladdierer. Eine größere Zahl von Volladdierern oder größere Volladdierer würden mehr als 32 Funktionseinheiten und einen größeren Instruktionsspeicher benötigen. Um die VHM zu einer einsetzbaren rekonfigurierbaren Hardware zu machen, muss die VHM ebenfalls mit *coarse grain* Funktionseinheiten ausgestattet werden. Dies hätte nicht nur den Vorteil, dass die Konfigurationsdaten kompakter und die VHM effizienter gestaltet werden könnte, sondern dass auch eine Erhöhung der Verarbeitungsgeschwindigkeit erzielt werden würde, da die Längen der kritischen Pfade deutlich kürzer wären. Durch die dadurch resultierende kleinere Anzahl von notwendigen Funktionseinheiten wäre auch der Kommunikationsaufwand innerhalb der VHM wesentlich geringer.

Die Kommunikation innerhalb der VHM ist ebenfalls stark verbesserungsbedürftig. Das momentane Konzept, in dem jede Funktionseinheit auf alle Daten zugreifen kann bewirkt, dass mehr

hardwareseitige Ressourcen für die Kommunikation verwendet werden als für die Berechnung selbst. Es lohnt sich somit die Kommunikation genauer zu analysieren, ob es nicht bessere Lösungen gibt die Funktionseinheiten mit Daten zu versorgen, ohne aber das Berechnungsverhalten der VHM nachteilig zu beeinflussen.

Auch bzgl. der Rekonfigurierbarkeit ist die VHM den anderen Ansätzen unterlegen. Um die allgemeine Lauffähigkeit der VHM zu testen, ist eine statische Rekonfiguration mit nur einer Konfiguration ausreichend. Für praktische Anwendungsgebiete hingegen ist diese Art der Rekonfiguration nur bedingt brauchbar, da die Berechnung nicht während der Konfiguration unterbrochen werden sollte und das Halten von mehreren Konfigurationen einen effektiveren Arbeitsablauf ermöglichen würde (Multitasking). Weiterhin geschieht die Konfiguration mit Hilfe eines sehr komplexen Decoders, welcher die mit Abstand aufwendigste Komponente der VHM darstellt. Dieser Decoder ermöglicht nur die Konfiguration der ganzen VHM, nicht aber das gezielte Umkonfigurieren einzelner Teile der VHM, was im Falle einer Rekonfiguration Zeit kostet. Es sollte überprüft werden, ob der Decoder wirklich notwendig ist oder ob er durch eine andere Konfigurationsschnittstelle ersetzt werden kann. Die Breite des Konfigurationsbuses ist mit 32 Bit sehr klein und trägt damit ebenfalls dazu bei, dass die Rekonfiguration mehr Zeit in Anspruch nimmt als notwendig.

Um die VHM aber wirklich mit anderen rekonfigurierbaren Architekturen vergleichen zu können, ist eine funktionierende Implementierung dieser unabdingbar. Dass die prototypische Implementierung der VHM mit Hilfe eines FPGAs nicht die Effizienz und Größe erreicht wie die Produkte XPP (2.3.1 auf Seite 16), Quicksilver (2.3.2 auf Seite 21) und Morphosys (2.3.3 auf Seite 26) ist verständlich und soll vorerst auch nicht das Ziel sein. Die gegenwärtige, nicht synthesefähige Implementierung der VHM [2] setzt auf die Strategie, alle Funktionseinheiten zusammen in einer Zeile in die VHM platzieren. Dies bedeutet, dass alle Komponenten geändert werden müssen, falls die Anzahl der Funktionseinheiten erhöht oder verringert würde.

3.1.1 Lösungsansätze

Granularität

Um die VHM von einer *fine grain* zu einer *coarse grain* Architektur zu überführen wäre es notwendig, komplexere Operationen wie Multiplikation, Addition, Subtraktion, Division, aber auch Multiplexer und Vergleichsoperationen zu implementieren, welche mit einer größeren Datenbreite als einem Bit arbeiten, z.B. 8, 16 oder 32 Bit. Am zweckmäßigsten ist die Implementierung von 32 Bit Operationen, da bei der notwendigen Implementierung von Festkommazahlen mit einer ausreichenden Genauigkeit der Wertebereich bei 8 bzw. 16 Bit Operationen unbrauchbar werden würde. Der Wertebereich wird, aufgrund der Verwendung von 10 Bit für die Darstellung der Nachkommastellen, von -2^{21}bis $2^{21} - \frac{1}{2^{10}}$ reichen. Die meisten der oben genannten Operationen sind bereits in den heutigen FPGAs als Komponenten enthalten oder es werden effektive Implementierungen für diese angeboten. Nur die Division fällt hier aus dem Rahmen. Da für die Division weder fest implementierte Komponenten, noch eine Implementierung mit hinreichend geringem Ressourcenverbrauch für die FPGAs von Xilinx existieren, wurde nur die Division mit Konstanten

erlaubt, was einer Multiplikation mit dem Kehrwert der Konstanten entspricht.

Die *fine grain* Elemente (Logik-Funktionseinheiten) sollen ebenfalls weiter enthalten bleiben, um mit ihrer Hilfe gegebenenfalls die *coarse grain* Elemente (RT-Funktionseinheiten) zu steuern, auch wenn das in der neuen Version noch nicht der Fall seien wird.

Implementierung

Es ist gelungen, die Implementierung der VHM von Nick Bierwisch [2] mit einigen Änderungen synthese- und auch lauffähig zu bekommen. Leider waren Teile des VHDL-Code nicht standardkonform, so dass mit der Verwendung neuerer Entwicklunswerkzeuge[1] eine Neuimplementierung notwendig gewesen wäre, welche aufgrund einer neuen Architektur sowieso erforderlich war. Das Ziel war es eine neue, flexibele Architektur, welche *fine* und *coarse grain* Funktionseinheiten[2] beinhaltet und in der Lage ist DSP-Aufgaben zu bewältigen, zu implementieren.

Um die VHM besser zu strukturieren und auch den Kommunikationsaufwand zu reduzieren (siehe nächster Abschnitt), werden Funktionseinheiten in sogenannten SubVHMs zusammengefasst (16 Logik-FUs in einer Logik-SubVHM und eine RT-FU in einer RT-SubVHM). Jede dieser SubVHMs ist für sich selbständig lauffähig und agiert unabhängig von den anderen SubVHMs. Es lassen sich beliebig viele SubVHMs (lediglich durch die Menge der zur Verfügung stehenden Ressourcen begrenzt) in einer VHM zusammenfassen. Die Funktionseinheiten sind in einer Zeile zusammengefasst, so dass alle Funktionseinheiten ihre Berechnungen parallel ausführen. Innerhalb einer SubVHM kann jede Funktionseinheit auf die Ergebnisse der anderen Funktionseinheiten zugreifen. Der Decoder für die Konfiguration ist durch eine externe Konfigurationsschnittstelle ersetzt worden.

Kommunikation

Die Kommunikation ist eine zentrales Probleme der meisten Computersysteme, welche mit mehr als einem Prozessor ausgestattet sind. Es muss immer auf die Aktualität der Ergebnisse geachtet werden, mit denen Berechnungen durchgeführt werden. Das große Problem der VHM ist, dass momentan alle Funktionseinheiten auf alle Register zugreifen können müssen, um die eigene Berechnung mit aktuellen Werten versorgen zu können. Dies resultiert in einem sehr großen Kommunikationsnetzwerk, welches durch eine Vielzahl von großen Multiplexern realisiert wird. Durch das Zusammenfassen von mehreren Funktionseinheiten zu einer SubVHM verringert sich der Kommunikationsaufwand[3] (siehe 4.1.1).

[1] Xilinx Projectnavigator > 5.1

[2] *fine grain* Elemente werden für die Logikebene verwendet und die RT-Ebene wird duch *corase grain* Elemente realisiert

[3] Es besteht der Vedacht, dass die Kommunikation noch wesentlich kleiner seien könnte (siehe 3.2.1/Kommunikation).

Rekonfiguration

Um die Konfiguration der VHM variabler zu gestalten, wurde der Decoder durch eine Konfigurationsschnittstelle ersetzt, welche das Laden von Konfigurationsdaten erheblich vereinfacht. War es vorher nur möglich eine Konfiguration im Ganzen einzulesen, so ist es nun möglich, gezielt einzelne Komponenten der VHM umzukonfigurieren. Hierfür müssen lediglich die Konfigurationsdaten angelegt und die Zielkomponente angegeben werden. Die VHM empfängt diese Konfigurationsanforderung und leitet sie an die spezifizierte Komponente weiter. Damit wird der VHM die Decodierung der Konfigurationsdaten abgenommen und an die Komponente abgegeben, welche sich der VHM bedient. Auch wenn die neue Implementierung der VHM nur eine Konfiguration auf einmal erlaubt, so ist durch die Verwendung von BlockRAMs ausreichend Platz für weitere Konfigurationen vorhanden und die Implementierung einer VHM mit mehreren Konfigurationen ohne größeren Aufwand möglich.

3.2 Die Schwachpunkte des VHBC-Compilers

Der VHBC-Compiler in seiner vor dieser Arbeit vorliegenden Form unterstützte ausschließlich VHDL-Code, welcher mit der SIMPRIM-Bibliothek von Xilinx konform war. Es handelt sich bei dieser VHDL-Bibliothek nicht um eine standardisierte Bibliothek, sondern um eine von Xilinx zusammengestellte Menge von Komponenten, mit deren Hilfe die Simulation einer Schaltung ermöglicht wird. Dadurch ergab sich eine Abhängigkeit von einem Hersteller, da der VHBC-Compiler nur mit der Ausgabe der Entwicklungswerkzeuge von Xilinx lauffähig war. Desweiteren bietet diese Bibliothek nicht die Möglichkeit der Einführung von Komplexeren Komponenten wie Mulziplierern, Addierern etc. und ist somit für die neue VHM nur noch bedingt brauchbar.

Ein weiterer Schwachpunkt ist, dass der VHBC-Compiler die eingelesene Netzliste nicht optimiert. Da die meisten Synthesetools auf eine bestimmte Zielarchitektur hin optimieren und aus diesem Grund einige Operationen favorisieren, während sie andere weg lassen, könnte eine Optimierung der Netzliste und die Ausnutzung aller in der VHM implementierten Operationen eine erheblichen Geschwindigkeitssteigerung bewirken.

Eine Untersuchung aller Teile des VHBC-Compilers auf ihre Notwendigkeit und Verwendbarkeit mit der neuen Architektur der VHM ist ohnehin unerlässlich. Die in Abschnitt 3.1.1 angesprochene Vermutung, dass der Kommunikationsaufwand innerhalb der VHM wesentlich geringer seien könnte fällt ebenfalls in das Aufgabengebiet des Compilers, da dieser für die Strukturierung und Aufteilung des VHBC-Codes verantwortlich ist.

3.2.1 Lösungsansätze

Eingabeformat

Um den VHBC-Compiler herstellerunabhängig zu gestalten und ihm die Verarbeitung komplexerer Komponenten zu ermöglichen, sollte der Compiler um ein unabhängiges, weit verbreitetes und einfach zu verarbeitendes Eingabeformat erweitert werden. Die Entscheidung fiel hierbei auf das EDIF-Format (siehe Abschnitt 2.1). Es ist ein unabhängiges Format, welches weder an einen speziellen Hersteller, noch an eine bestimmte Hardware gebunden ist. Darüber hinaus bietet es die Möglichkeit, beliebig komplexe Komponenten in jeder verwendbaren Sichtweise (Verhaltensbeschreibung, strukturelle Beschreibung, grafische Beschreibung etc.) zu beschreiben. Auch wenn das EDIF-Konsortium sich aufgelöst hat, so stellt EDIF ein sehr mächtiges und zukunftsträchtiges Eingabeformat dar.

Netzlistenoptimierung

Da die VHM nicht für so viele Logikoperationen Platz bietet, wie auf einem FPGA Platz finden würden, ist eine Optimierung der Netzliste ein guter Ansatz, dieses Defizit zu verkleinern. Dabei sollten redundante Komponenten zusammengefasst und logische Operationen optimiert werden. Bei der Optimierung handelt es sich um Logikoptimierungen, welche auf Operationen mit maximal zwei Operanden angewendet werden, z.B. $(A \wedge B) \vee (A \wedge C))$ wird zu $A \wedge (B \vee C)$.

Kommunikation

Bei der Analyse der Kommunikation war empirisch festzustellen, dass die Anzahl der notwendigen Kommunikationen wesentlich geringer ist. Die Ausgangslage stellt sich so dar, dass eine SubVHM, welche $n > 2$ Eingänge besitzt, sich für ihre n Funktionseinheiten maximal n neue Werte von anderen SubVHMs besorgen müsste. Die Vermutung ist, dass durch geeignete Vertauschung von Instruktionen und Eingangswerten zwischen den SubVHMs sich die Anzahl der Kommunikationen zwischen den SubVHMs von maximal n auf maximal $log(n)$ Kommunikationen reduziert (Abb. 3.1). Es wurde zwar bis jetzt kein Gegenbeispiel gefunden, doch wird diese Optimierung erst implementiert werden, wenn ihre Korrektheit einwandfrei bewiesen ist.

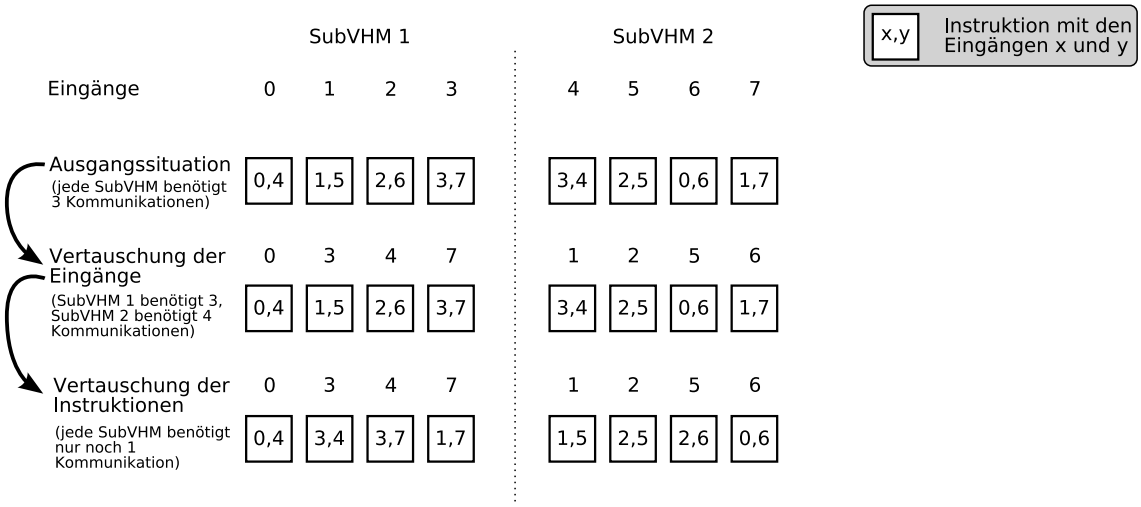


Abbildung 3.1: Vertauschung der Eingänge und der Instruktionen, um die Anzahl der notwendigen Kommunikationen zu reduzieren am Beispiel von zwei SubVHMs mit $n = 4$.

3.3 Zusammenfassung

Durch die Verbesserungen ergibt sich folgende Klassifizierung:

- *Granularität=mixed grain*: Es existieren sowohl *coarse grain,* als auch *fine grain* Funktionseinheiten (RT- und Logikebene). Die *coarse grain* Funktionseinheiten arbeiten mit 32 Bit Festkommazahlen (zehn Bit für die Nachkommastellen).
- *Rekonfigurierbarkeit=statisch*: Jede Rekonfiguration unterbricht die VHM in ihrer Arbeit. Die Abarbeitung wird erst wieder aufgenommen, wenn die Konfiguration abgeschlossen ist. Der Übergang zu dynamischer Rekonfiguration ist kein Problem, wenn zeitgleich die Tiefe der Programmierung auf mindestens 2 erhöht wird[4].
- *Tiefe der Programmierung=1*: Auch wenn jede FU einen Instruktionsspeicher für 16 Instruktionen besitzt, so gibt es nur eine, in der VHM gehaltene Konfiguration. Die Erhöhung der Programmiertiefe ließe sich ohne größeren Aufwand realisieren.
- *Schnittstelle=local*: Momentan ist die VHM über den PLB[5] mit einem, auf dem FPGA vorhandenen PowerPC, verbunden, welche die VHM mit Daten versorgt. Erstellt wurde das Gesamtsystem mit Hilfe des EDK von Xilinx.
- *Berechnungsmodel=MIMD*: Die Aufbau der VHM und des VHBC sind für MIMD-Berechnungen ausgelegt.

Die VHM ist durch die angestrebten Verbesserungen auf dem besten Weg eine leistungsstarke und flexible rekonfigurierbare Hardware zu werden. Dass sie nicht so leistungsstark ist wie die anderen vorgestellten Ansätze kommt daher, das diese einerseits in fester Hardware implementiert sind und andererseits wesentlich mehr Arbeit in sie investiert wurde.

[4] noch nicht implementiert
[5] Processor Local Bus

Kapitel 4

Implementierung

Das Ziel dieser Diplomarbeit ist eine Optimierung und Erweiterung des VHBC-Compilers und die Implementierung der VHM mit Hilfe von VHDL. Aufbauend auf den vorangegangenen Arbeiten und neuen Ideen von Professor Kebschull [17] ergab sich eine neue, vom Prinzip her einfache und leicht skalierbare VHM. So besteht die neue VHM nicht mehr aus einer großen Anzahl von kleinen, unabhängigen Funktionseinheiten, sondern aus einer geringeren Menge von selbständigen SubVHMs. Darüber hinaus wurde der übermäßig komplexe Decoder, der für das Einlesen der Konfigurationsdaten verantwortlich war, entfernt und durch eine einfache Konfigurationsschnittstelle ersetzt. Eine weitere Änderung ist die Aufnahme von EDIF als Quellformat für den Compiler. In den beiden folgenden Abschnitten werden die Änderungen und Neuerungen erläutert.

Vorausgehend soll jedoch beschrieben werden, wie die Struktur des VHM-Arbeitsprozesses ist. Im Prinzip handelt es sich um eine Erweiterung des in Kapitel 2.4 beschriebenen Arbeitsablaufs. Die einzige gravierende Veränderung betrifft die Konfiguration der VHM. Diese geschieht nun nicht mehr aus eigener Kraft, indem ein eingebauter Decoder die VHBC-Daten verarbeitet, sondern wird von außen gesteuert. Die Konfiguration geht so vonstatten, dass der VHM signalisiert wird, wenn sie Konfigurationsdaten übernehmen kann, welche daraufhin eingelesen werden und das nach außen gemeldet wird. Die Berechnung ist auch in dieser Version in Macro- und Microzyklen unterteilt. Ein Macrozyklus bezeichnet hierbei das einmalige, komplette Durchlaufen der im VHBC beschriebenen Schaltung. Sie läuft so ab, dass zuerst die für die einzelnen Komponenten notwendigen Werte beschafft werden (Kommunikation), um anschließend die Berechnung durchzuführen. Abbildung 4.1 zeigt noch einmal eine abstrakte Darstellung des Arbeitsablaufs.

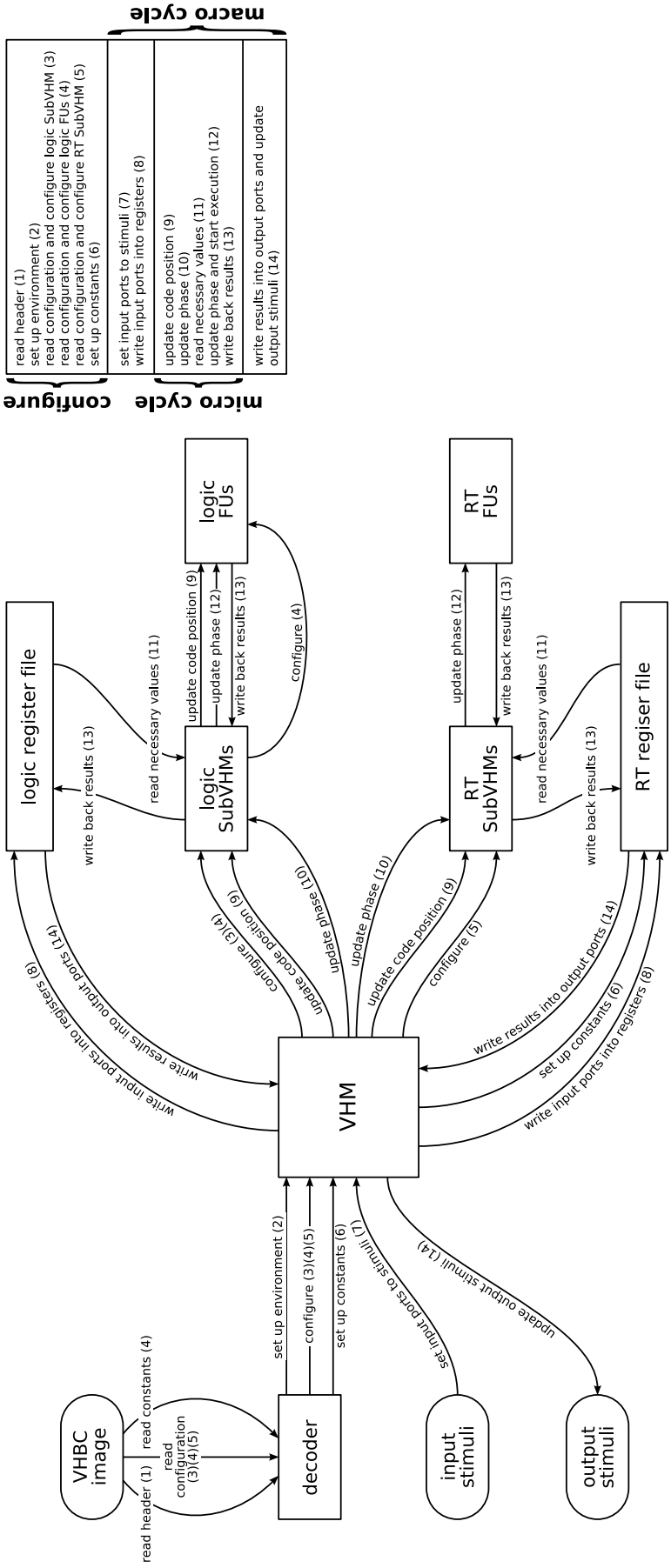


Abbildung 4.1: Schematische Darstellung des Arbeitsablaufs der VHM. Zuerst erfolgt die Konfiguration (1 bis 5). Anschließend durchläuft die VHM immer wieder den Zyklus: Eingabewerte einlesen (7,8), sooft den Mikrozyklus durchlaufen wie es die Berechnung erfordert (9 bis 13), Ergebnisse ausgeben (14)

4.1 Die Virtuelle Hardware Machine

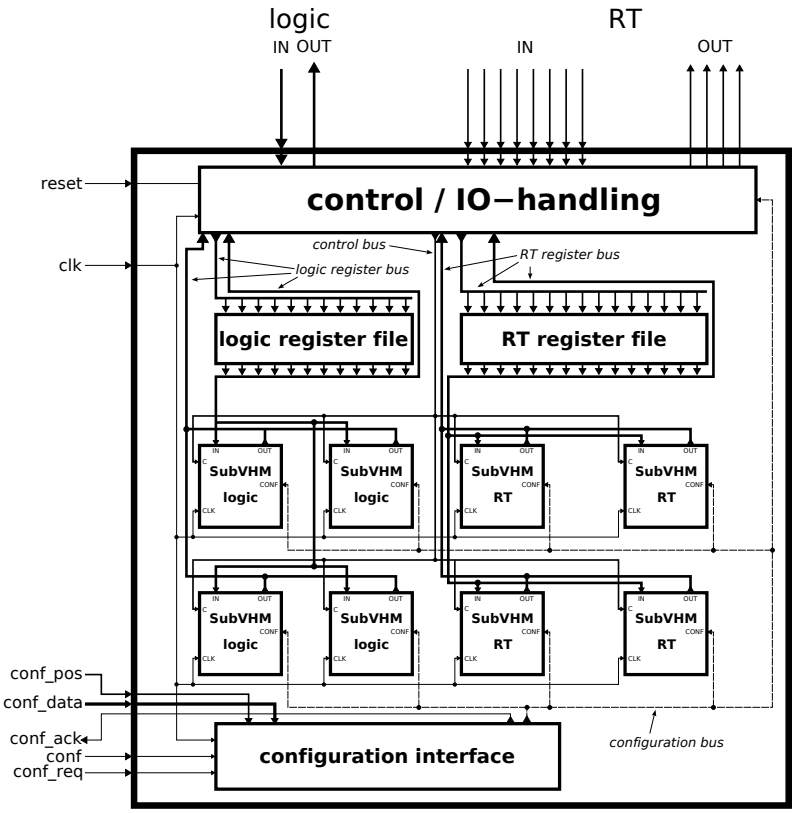


Abbildung 4.2: Die VHM, die alles vereinende Komponente. Auf diesem Bild enthält die VHM nur 4 RT-SubVHMs im Gegensatz zu 16 RT-SubVHMs in der realen Implementierung.

Die VHM (Abb. 4.2) ist eine Maschine, welche entwickelt wurde, um die Funktionalität eines, an anderer Stelle entwickelten Hardwaredesigns zu übernehmen. Um das zu gewährleisten ist es notwendig, eine breite Palette von Standardoperationen, welche am häuftigsten in Hardwaredesigns vorkommen, zu implementieren. Obwohl sich mit Hilfe der bereits vorher implementierten Logikoperationen jede Schaltung realisieren ließe, ist dieser Ansatz aber ungeeignet, da die Implementierung einiger komplexer Operationen sehr große Ausmaße annehmen kann. Wesentlich besser ist es in diesem Fall auf bereits vorhandene Komponenten zurückzugreifen und somit die Komplexität und Größe des VHBC und der VHM zu reduzieren. Im Detail heißt das, dass eine neue Ebene von Instruktionen eingeführt wurde, welche in der Lage ist, Schaltungen mit komplexeren Operationen zu realisieren. Es handelt sich hierbei um Operationen auf RT-Ebene, die nicht auf einzelne Bitwerte angewendet werden, sondern auf einer Datenbreite von 32-Bit arbeiten.

Um die Verständlichkeit zu verbessern, werden nachfolgend Logik- und RT-Ebene getrennt behandelt. Das verdeutlicht ebenfalls, dass die beiden Ebenen unabhängig voneinander betrieben werden können. In späteren Versionen der VHM soll sich das noch ändern, um den Ebenen die Möglichkeit zu geben auf die andere Ebene Einfluss nehmen zu können.

4.1.1 Logikebene

Der Funktionsumfang der Logikebene ist unverändert. Lediglich die Zuweisung von Operationscodes zu den Operationen ist umgestalltet worden. Waren diese bis jetzt nur durchnummeriert, so repräsentiert der Operationscode einer Operation jetzt ihre logische Wertetabelle. Diese Darstellung ermöglicht eine effiziente Implementierung in Hardware, da der Operationscode selbst als eine Art LUT zu verstehen ist. Abbildung 4.3 verdeutlicht dies am Beispiel einer AND-Operation und Tabelle 4.1 listet noch einmal alle implementierten Logikoperationen auf. Die Operationscodes für die NOP- und die CMOV-Instruktion[1] fallen nicht in das eben beschriebene Bildungsschema und wurden so gewählt, da keine logische Operationen mit diesen Wertetabellen existieren, welche nicht schon in dieser Art vorhanden wären.

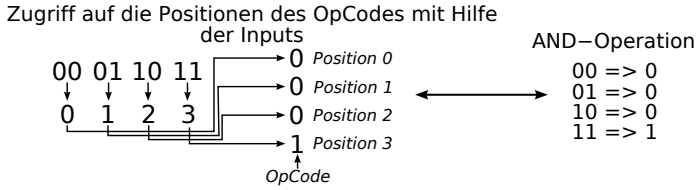


Abbildung 4.3: Beispiel einer AND-Operation. Der Operationscode stellt gleichzeitig die Wertetabelle einer Operation dar.

Tabelle 4.1: Operationscodes der VHM für die Logikebene

OpCode		Mnemonic	mathematischer Ausdruck (a,b=Inputs;c=Output)
0_{10}	0000_2	ZERO	$c = false\,(0)$
1_{10}	0001_2	AND	$c = a \wedge b$
2_{10}	0010_2	NIMP	$c = \neg(\neg a \vee (a \wedge b))$
3_{10}	0011_2	MOV	$c = a$
6_{10}	0110_2	XOR	$c = (\neg a \wedge b) \vee (a \wedge \neg b)$
7_{10}	0111_2	OR	$c = a \vee b$
8_{10}	1000_2	NOR	$c = \neg(a \vee b)$
9_{10}	1001_2	EQ	$c = (a \wedge b) \vee (\neg a \wedge \neg b)$
10_{10}	1010_2	CMOV	$c = \begin{cases} b \text{ if } a \text{ true} \\ c \text{ otherwise} \end{cases}$
11_{10}	1011_2	NOP	$c = c$
12_{10}	1100_2	NOT	$c = \neg a$
13_{10}	1101_2	IMP	$c = \neg a \vee (a \wedge b)$
14_{10}	1110_2	NAND	$c = \neg(a \wedge b)$
15_{10}	1111_2	ONE	$c = true\,(1)$

Alle in Tabelle 4.1 aufgelisteten Operationen sind in den Logikfunktionseinheiten (Abb. 4.5) implementiert. Diese stellen die für die Berechnung verantwortlichen Elemente der Logikebene dar und sind in der Lage, parallel und unabhängig voneinander zu arbeiten. Das Einzige was sie dazu benötigen sind zwei Eingangswerte und ein Operationscode. Um nicht alle Funktionseinheiten auf

[1] NOP=No OPeration; CMOV=Conditional MOVe

einmal in die VHM zu packen und um die VHM besser zu strukturieren, wurden jeweils 16 Funktionseinheiten in einer SubVHM (Abb. 4.4) zusammengefasst. Das dadurch ebenfalls der Kommunikationsaufwand innerhalb der VHM reduziert wird, ist ein schöner Nebeneffekt und wird später näher erklärt, nachdem der Aufbau einer SubVHM für die Logikebene genauer erläutert wurde.

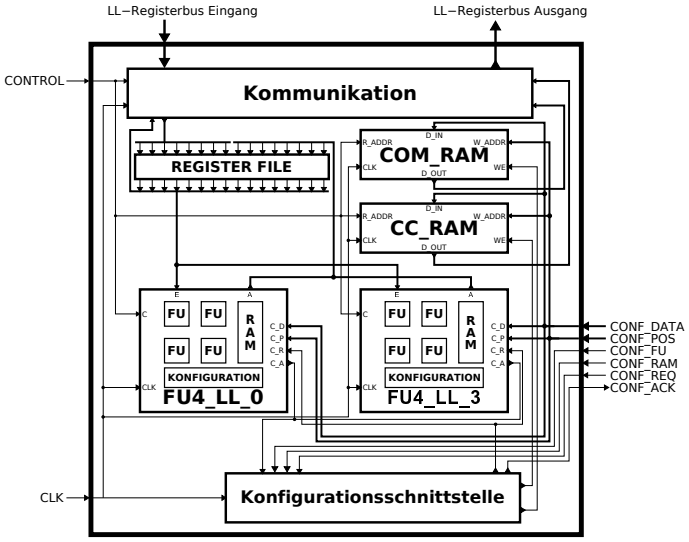


Abbildung 4.4: Eine SubVHM der Logikebene. Die hier abgebildete SubVHM enthält für die bessere Übersicht nur 8 statt 16 Funktionseinheiten. Zu beachten ist das Zusammenfassen von vier Funktionseinheiten zu einem Verbund.

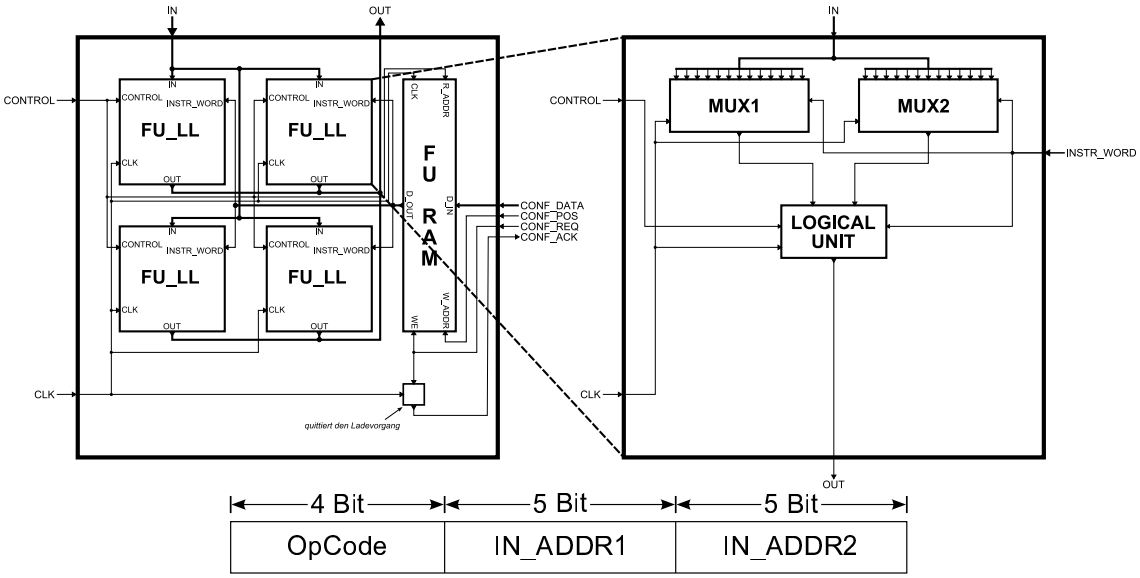


Abbildung 4.5: Eine Vierergruppe von Funktionseinheiten (oben links), eine einzelne Funktionseinheit (oben rechts) der Logikebene und das Instruktionswort einer Funktionseinheit (unten)

Eine SubVHM für die Logikebene besteht, wie beschrieben, aus 16 Logikfunktionseinheiten. Die eigentliche Architektur der SubVHM für die Logikebene sieht kein weiteres Zusammenfassen von Funktionseinheiten vor, wie es in Abbildung 4.5 oben links dargestellt ist. Dieses Zusammenfassen

von vier Funktionseinheiten zu einem Verbund entstand aus der zweierlei Gründen. Erstens waren die Ressourcen auf dem Chip[2], welcher zur Verfügung stand, begrenzt, so dass mit der Menge der verwendeten BlockRAMs hausgehalten werden musste. Zweitens ergibt dieses Zusammenfassen von vier Funktionseinheiten eine Konfigurationsbreite von 56 Bit (je 14 Bit), was eine bessere Ausnutzung des 64-Bit breiten Konfigurationsbusses erlaubt. Dadurch wird ebenfalls die Konfigurationszeit erheblich verkürzt. Jede dieser Funktionseinheiten besitzt ein Ausgaberegister mit einer Breite von einem Bit. Diese Ausgaberegister repräsentieren die oberen 16 Bit eines 32 Bit breiten Registerfiles, welches die SubVHM für die Zwischenspeicherung von Eingabewerten und selbst berechneten Werten verwendet. Für die Eingabewerte der 16 Eingänge stehen dabei die unteren 16 Bit des Registerfiles zur Verfügung. Jede Funktionseinheit ist in der Lage auf, jedes der Register des Registerfiles lesend zuzugreifen, während es aber nur in das, seiner Position entsprechende Ausgaberegister schreiben darf. Dies bedeutet, dass die erste Funktionseinheit in Register 16 schreiben darf, die zweite in Register 17 usw. Die Ausgaberegister repräsentieren ebenfalls die 16 Ausgänge der SubVHM. Jede SubVHM ist in der Lage, die Ausgaberegister der anderen SubVHMs zu lesen. Hierfür besitz die VHM ein Registerfile, welches für jeden Ausgang einer SubVHM ein Register besitzt und jede SubVHM somit zusammenhängend 16 Register belegt. Falls eine Funktionseinheit einer SubVHM das Ergebnis einer Funktionseinheit einer anderen SubVHM benötigt, so kopiert die SubVHM das benötigte Ergebnis aus dem Registerfile der VHM in eines ihrer 16 Eingangsregister und macht es somit der Funktionseinheit zugänglich. Stark vereinfacht funktioniert die Kommunikation in einer Ob-Von wem-Welches Register-Art. Hierfür dienen die Daten aus dem *com*[3] *word* (48 Bit) und dem *rn*[4] *word* (64 Bit) (Abb. 4.6). Im Detail heißt das, dass die ersten 16 Bit des *com word* dazu verwendet werden, um zu entscheiden, ob dem, seiner Position entsprechenden Eingangsregister der SubVHM ein externer (bezogen auf die SubVHM) Wert zugewiesen werden soll, was der Fall ist, wenn das entsprechende Bit auf 1 gesetzt ist. Die restlichen 32 Bit des *com word* enthalten die Nummern der SubVHMs, von welchen die Werte geholt werden sollen. Das *rn word* enthält die Nummern der Register die den benötigten Wert enthalten. Hierbei gehören gleiche Positionen in den einzelnen Konfigurationensdaten zusammen, d.h. Bit n der ersten 16 Bit des *com word* gehört mit den Bits ($2 * n$,$2 * n + 1$) der restlichen 32 Bit des *com word* und den Bits ($4 * n$,$4 * n + 3$) des *rn word* zusammen. Alle maximal 16 Kommunikationen werden parallel ausgeführt.

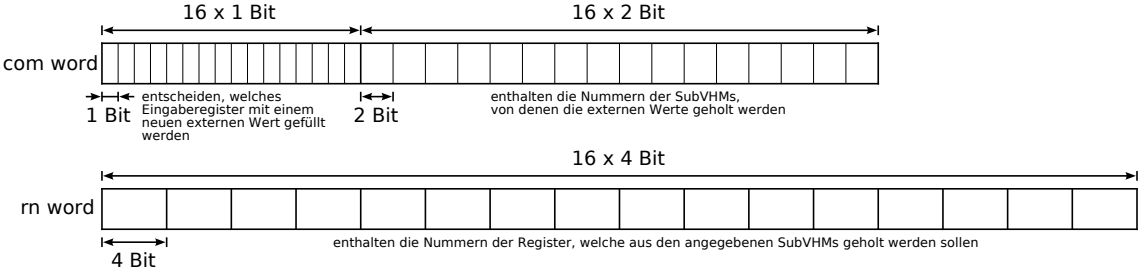


Abbildung 4.6: Die für die Kommunikation wichtigen Datenworte *rn word* und *com word*

[2] Virtex2P xc2vp30-7ff896

[3] *com*=communication

[4] *rn*=register number

Damit stellt die SubVHM nicht nur eine Komponente dar, welche die strukturelle Übersicht der VHM verbessert, sondern sie übernimmt auch die Verantwortung für die Kommunikation zwischen den Funktionseinheiten. Die vorher angesprochene Reduzierung des Kommunikationsaufwandes kommt nun dadurch zustande, dass nicht mehr jede Funktionseinheit auf alle Ausgaberegister aller anderen Funktionseinheiten zugreifen können muss. Bei 256 Funktionseinheiten mit je zwei Eingängen würde das bedeuten, dass jede Einheit zwei 256-1 Multiplexer für seine Eingänge benötigen würde, wohingegen in der Variante mit den SubVHMs jede Einheit lediglich zwei 32-1 Multiplexer benötigt. Diese Einsparung wird teilweise aber wieder dadurch aufgebraucht, dass jede SubVHM 16 240-1 Multiplexer an seinen Eingängen benötigt (die Ausgänge der anderen SubVHMs). Würden diese Multiplexer auf einem FPGA mit Hilfe von 2-1 Multiplexern realisiert werden müssen, so hätte man mit dem neuen Ansatz 61696 2-1 Multiplexer eingespart[5]. Der größere Vorteil liegt aber in der Aufteilung der Kommunikation, so dass nicht mehr eine Komponente alles erledigen muss, sondern sich SubVHM und Funktionseinheit die Arbeit teilen.

Nachdem die SubVHM die notwendigen Kommunikationen durchgeführt hat, beginnen die Funktionseinheiten mit ihrer Berechnung. Alles was sie hierfür benötigen ist der Zugriff auf das 32 Bit Registerfile und je ein 14 Bit breites Instruktionswort (Abb. 4.5 unten). Abbildung 4.7 zeigt den VHDL-Code, welcher die Berechnung beschreibt. Zeile 13 zeigt die bereits beschriebene Verwendung des Operationscodes als eine Art LUT. Das Berechnungsergebnis wird in das, der Funktionseinheit zugewiesene Ausgaberegister geschrieben.

```
1     opCode := instr_word(0 to 3);
2     in1 := inRegs(CONV_INTEGER(instr_word(4 to 8)));
3     in2 := inRegs(CONV_INTEGER(instr_word(9 to 13)));
4
5     if opCode = "1010"            -- CMOV
6     then
7           if in1 = '1'
8           then
9                 result <= in2;
10          end if;
11    elsif opCode /= "1011"        -- not NOP, opCode as LUT
12    then
13          result <= opCode(CONV_INTEGER(in1&in2));
14    end if;
```

Abbildung 4.7: VHDL-Code der Berechnung auf Logikebene

Die Konfigurationsdaten, welche die Funktionalität der SubVHM und der einzelnen Funktionseinheiten definieren, werden in sechs verschiedenen Speichern aufbewahrt. Zwei davon verwendet die SubVHM für die Kommunikation und die restlichen dienen dazu, die Instruktionsworte für die Funktionseinheiten zu Verfügung zu stellen, wobei immer vier Funktionseinheiten wie be-

[5] Für die Realisierung eines 256-1 Multiplexers werden 255 2-1 Multiplexer benötigt. Wenn jede der 256 Funktionseinheiten zwei solche 256-1 Multiplexer hätte, so würden insgesammt 130560 2-1 Multiplexer benötigt werden. Verwendet jede Funktionseinheit nut zwei 16-1 Multiplexer (15 2-1 Multiplexer), so werden innerhalb der SubVHM nur 480 2-1 Multiplexer benötigt, plus die 240-1 Multiplexer an den Eingängen der SubVHMs, von denen jede 16 Stück besitzt (240-1 Multiplexer benötigt 239 2-1 Multiplexer, 16 Mal pro SubVHM). Damit benötigt jede SubVHM noch einmal 3824 2-1 Multiplexer, was zu einer Gesammtzahl an 2-1 Multiplexern von 68864 führt.

reits beschrieben an einen Speicher angeschlossen sind. Die Konfiguration selbst wird von außen gesteuert. Hierfür dienen folgende Ein- bzw. Ausgangssignale (vgl. Abb. 4.4)

- *conf_pos* (4 Bit Eingang): spezifiziert die Speicheradresse, welche beschrieben werden soll
- *conf_ram* (2 Bit Eingang): spezifiziert den Speicher, welcher beschrieben werden soll (FU-, RN- oder COM-RAM)
- *conf_fu* (2 Bit Eingang): spezifiziert die Vierergruppe von Funktionseinheiten, welche konfiguriert werden sollen
- *conf_data* (64 Bit Eingang): stellt den Eingang dar, an welchem die Konfigurationsdaten anliegen
- *conf_req* (1 Bit Eingang): startet die Übernahme der Konfigurationsdaten
- *conf_ack* (1 Bit Ausgang): quittiert die Übernahme der Konfigurationsdaten

Um eine SubVHM zu konfigurieren, müssen zuerst die Konfigurationsdaten an *conf_data* angelegt werden. Der Wert von *conf_ram* entscheidet, ob eine Vierergruppe von Funktionseinheiten (00), der COM-RAM (10) oder der RN-RAM (01) konfiguriert werden. Soll eine Funktionseinheitengruppe konfiguriert werden, so gibt *conf_fu* an welche. Der Wert von *conf_pos* spezifiziert die zu konfigurierende Adresse, welcher der Nummer eines Mikrozyklus entspricht. Wird nun *conf_req* auf 1 gesetzt, so beginnt die Übernahme der Konfigurationsdaten. Im Fall der Konfiguration eines Speichers der SubVHM führt eine UND-Verknüpfung von *conf_req* und *conf_ram(0)* bzw. *conf_ram(1)* zum Setzen des 'write enable'-Bits des COM- bzw. des RN-RAMs, womit die an den Eingängen der beiden Speicher anliegenden Konfigurationsdaten aus *conf_data* übernommen werden. Gleichzeitig wird *conf_ack* auf 1 gesetzt und signalisiert damit nach aussen, dass die Konfigurationsdaten übernommen worden sind.

Die Konfiguration einer Funktionseinheitengruppe ist noch einen Schritt länger. Wird hier *conf_req* auf 1 gesetzt, so wird dieses an die in *conf_fu* angegebene Funktionseinheitengruppe weitergeleitet. Bekommt eine Vierergruppe eine solche, durch das Setzten von *conf_req* signalisierte Konfigurationsanfrage, so übernimmt sie umgehend die Konfigurationsdaten und quittiert dies mit einer 1 in *conf_ack*. Erst wenn die Übernahme der Konfigurationsdaten durch *conf_ack* bestätigt worden ist, kann mit der Konfiguration fortgeschritten werden. Im letzten Schritt wird *conf_req* wieder eine 0 zugewiesen, was die SubVHM bzw. die Funktionseinheitengruppe veranlasst, *conf_ack* ebenfalls auf 0 zurückzusetzen. Wie die externen Signale genau angeschlossen sind, kann in den Abbildungen 4.4 und 4.5 noch einmal genauer betrachtet werden.

4.1.2 RT-Ebene

Die Implementierung von Operationen auf RT-Ebene ist für die VHM vollkommen neu. Da eine Realisierung dieser Operationen mit Hilfe logischer Gatter nicht effektiv wäre (hoher Verbrauch an Ressourcen, erhebliche Verlängerung des kritischen Pfades) und auf den meisten modernen programmierbaren Chips die entsprechenden Operationen verfügbar sind (z.B. verfügt der Xilinx XC2VP30 über 136 18x18 Multiplizierer [14]), ist eine direkte Implementierung die sinnvollste Lösung. Bei den neuen Operationen handelt es sich um:

- Rechenoperationen: Addition (ADD), Subtraktion (SUB), Multiplikation (MULT) und Division mit Konstanten (DIV)
- Operationen zu Flusssteuerung: Multiplexer (MUX), Vergleich (COMP, liefert einen boolschen Wert als Ergebnis)
- Operationen, welche für den Berechnungsablauf benötigt werden: Leeroperation (NOP), Verschiebeoperation (COMPMOV_LT, COMPMOV_GT)

Alle Operationen arbeiten mit vorzeichenbehafteten 32-Bit Werten. Lediglich die Multiplikation verfügt über zwei nur 18-Bit breite Eingänge. Die vier Bits, welche bei einer Multiplikation von zwei 18-Bit Werten über sind, werden verworfen. Die Multiplikation fällt hier aus der Rolle, da der für die Tests zur Verfügung stehende FPGA 18x18 Multiplikatoren direkt implementiert hat. Ungeachtet dessen ist es trotzdem möglich einen Multiplizierer zu implementieren, welcher auf 32-Bit Werten arbeitet, was sich aber in einem höheren Verbrauch an Ressourcen und in einem deutlichen Absinken der maximal möglichen Taktrate widerspiegelt. Im konkreten Fall bedeutet das eine Reduzierung der Taktrate von 170 auf 104 MHz und einem Verbrauch von vier statt einem 18x18 Multiplizierer für eine Multiplikation. Da es sich bei allen Operationen nicht um Gleitkommaoperationen handelt, gleichzeitig aber eine Implementierung von reiner Ganzzahlarithmetik nicht sehr praktikabel wäre, wurde sich für Festkommaberechnungen mit einem Verbrauch von 10 Bit für die Nachkommastellen entschieden. Das bedeutet, dass jede Zahl durch ihr 2^{10}-faches repräsentiert wird. Somit ergibt sich die kleinste, nicht negative Zahl ungleich Null aus $\frac{1}{2^{10}} = 0,0009765625$ und wird durch die 1 repräsentiert. Während es für die anderen Operationen keinen Unterschied macht, ob sie Ganzzahl- oder Festkommawerte bearbeiten, müssen die Ergebnisse der Multiplikationen angepasst werden. Da jeder Wert durch sein Vielfaches von 2^{10} repräsentiert wird, würde die Multiplikation der Werte $x = a * 2^{10}$ und $y = b * 2^{10}$ den Wert $x * y = a * 2^{10} * b * 2^{10} = a * b * 2^{20}$ ergeben, welcher um den Faktor 2^{10} zu groß ist. Aus diesem Grund muss das Ergebnis einer jeden Multiplikation um zehn Bit nach rechts verschoben werden. An dieser Stelle offenbart sich ein weiteres Problem mit der Multiplikation. Durch die Verwendung von zehn Bit für die Nachkommastellen ergibt sich für die Eingänge der Multiplikationen, basierend auf einem 18x18 Multiplikator nur noch ein effektiver Wertebereich von -128 bis 127,9990..., was bei realistischer Betrachtung keinen sehr brauchbaren Wertebereich darstellt. Aus diesem Grund wurden zwei Versionen getestet. Eine, welche die 'kleine' Multiplikation verwendet und dadurch schneller, aber auch eingeschränkter ist und eine, welche die vollen 32 Bit

für die Multiplikation verwendet, dadurch aber wesentlich langsamer ist. Genaueres zu den Tests wird in Kapitel 5 beschrieben.

Da der FPGA, welcher zur Verfügung stand, keine Dividierer implementiert hat, wurde lediglich die Division mit einer Konstanten realisiert. Eine Implementierung mit Hilfe der verbleibenden Ressourcen des FPGA wäre zwar möglich, aber um einiges zu groß gewesen[6].

Die Multiplexer und die Vergleichsoperationen sind für die Steuerung der Berechnung auf algorithmischer Ebene zuständig. Beim Multiplexer handelt es sich um einen 32 Bit 2-1 Multiplexer, dessen 1 Bit breiter *Select*-Eingang den dritten Eingang der Funktionseinheit darstellt. Die Vergleichsoperation implementiert eine $<$-Relation. Ihr Ergebnis schreibt sie in das niederwertigste Bit der Ausgabe. Multiplexer und Vergleichsoperation dienen z.B. dazu, die Berechnung einer Summe so zu beeinflussen, dass a) die Summe erst ausgegeben wird wenn sie vollständig berechnet wurde und b) die Summe nur bis zu einem bestimmten Wert gebildet wird und anschließen die Berechnung von Neuem beginnt. Abbildung 4.8 verdeutlicht das noch einmal grafisch am Beispiel einer Summe.

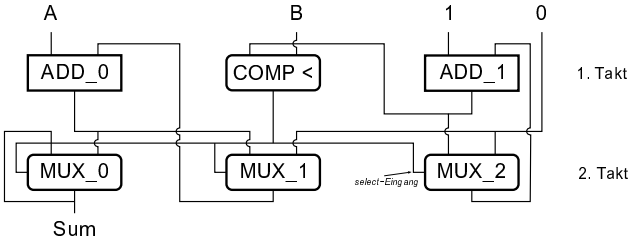


Abbildung 4.8: Berechnung der Summe $\sum_{0}^{B} A$

Die MOV und die NOP-Operationen stellen keine berechnenden Komponenten dar und greifen auch nicht direkt in die Berechnung ein. Sie dienen eher dem Datenfluss innerhalb des VHBC und tragen dafür Sorge, dass berechnete Werte, welche zu irgend einem späteren Zeitpunkt noch einmal benötigt werden, bis zu diesem Punkt gültig bleiben. Bei der MOV-Operation greschieht dies auf aktive Art und Weise, indem gezielt Werte verschoben, bzw. geholt und in das eigene Ausgaberegister geschrieben werden. Dabei wird einfach nur der am ersten Eingang anliegende Wert in das Ausgaberegister geschrieben. Die NOP-Operation wiederum funktioniert so, dass sie ihrem Namen alle Ehre macht und nichts tut, was bedeutet, dass sich der Ausgabewert der Funktionseinheit nicht ändert und somit gültig bleibt. In Tabelle 4.2 sind noch einmal alle Operationen der RT-Ebene aufgelistet. Die Vergabe der Operationscodes hat hier, im Gegensatz zur Logikebene, keinen tieferen Sinn und dient lediglich der Unterscheidung der Operationen.

[6] Ein Dividierer würde ungefähr 10% der Ressourcen belegen, was bei 16 Funktionseinheiten ein erhebliches Defizit bedeuten würde.

Tabelle 4.2: Operationscodes der VHM für die RT-Ebene

OpCode		Mnemonic	mathematischer Ausdruck (a,b,sel=Inputs; c=Output)
0_{10}	000_2	ADD	$c = a + b$
1_{10}	001_2	SUB	$c = a - b$
2_{10}	010_2	MUL	$c = a * b$
3_{10}	011_2	MUX	$c = \begin{cases} a \text{ if } sel = 0 \\ b \text{ if } sel = 1 \end{cases}$
4_{10}	100_2	COMPMOV_LT[7]	$c = \begin{cases} a \text{ if } a < b \\ b \text{ if } a \geq b \end{cases}$
5_{10}	101_2	COMP	$c = \begin{cases} 0 \text{ if } a < b \\ 1 \text{ if } a \geq b \end{cases}$
6_{10}	110_2	COMPMOV_GT[7]	$c = \begin{cases} a \text{ if } a > b \\ b \text{ if } a \leq b \end{cases}$
7_{10}	111_2	NOP	$c = c$

Wie auch in der Logikebene werden alle Operationen in einer Funktionseinheit (Abb. 4.9) implementiert. Die Funktionseinheiten arbeiten ebenfalls parallel und unabhängig voneinander. Im Gegensatz zur Logikebene beinhaltet aber eine SubVHM für die RT-Ebene (Abb. 4.10) nur eine einzige Funktionseinheit und ist ansonsten sehr einfach strukturiert. So sind die einzigen weiteren Komponenten in der SubVHM zwei 32-Bit und ein 1-Bit Eingangsregister für die Funktionseinheit sowie ein Konfigurationsspeicher. Auch die SubVHM für die RT-Ebene trägt dafür Sorge, dass die Funktionseinheit mit den notwendigen Eingabewerten versorgt wird. Dies geschieht, indem sie mit Hilfe des 15 Bit breiten Instruktionswortes (Abb. 4.11) die drei benötigten Eingabewerte in die internen Register überträgt. Bei diesen drei Werten handelt es sich um die zwei Eingänge, welche jede Operation besitzt und dem einen Bit breiten *Select*-Eingang des Multiplexers. Der *Select*-Eingang ist mit dem niederwertigsten Bit des dritten Wertes verbunden, welcher das Ergebnis einer Vergleichsoperation dastellt. Der Ausgabewert der Funktionseinheit wird der VHM übergeben und nicht in der SubVHM gespeichert.

Nachdem die Kommunikation abgeschlossen ist, beginnt die Berechnung. Hierbei wird mit Hilfe eines Multiplexers die auszuführende Operation ausgewählt, welche ihr Ergebnis dann in den Ausgang der Funktionseinheit schreibt. Dabei liegt am *Select*-Eingang des Multiplexers der momentane Operationscode an, welcher in den letzten drei Bit des Instruktionswortes der SubVHM (Abb. 4.11) steht. Das Ergebnis wird direkt an den Ausgang weitergeleitet und nicht innerhalb der SubVHM gespeichert. Im VHDL-Code ist nur das Verhalten der Berechnung beschrieben und es wird dem Synthesewerkzeug die Optimierung bzw. die Verwendung vorhandener Komponenten wie Multiplizierer oder Addierer überlassen. Die Berechnung von Addition und Subtraktion ist ganz normal. Bei der Multiplikation hingegen kommt der Umstand zum Tragen, dass es sich bei

[7] Wenn es nur darum geht einen Wert zuverschieben, so sind an Eingang a und b die selben Werte angelegt. Die MOV-Operation mit einem Vergleich zu koppeln hat den Vorteil, innerhalb eines Mikrozyklus einen Vergleich und die entsprechende Verschiebung durchführen zu können und hilft somit die zwei Mikrozyklen dauernde COMP-MUX-Kombination einzuspaaren.

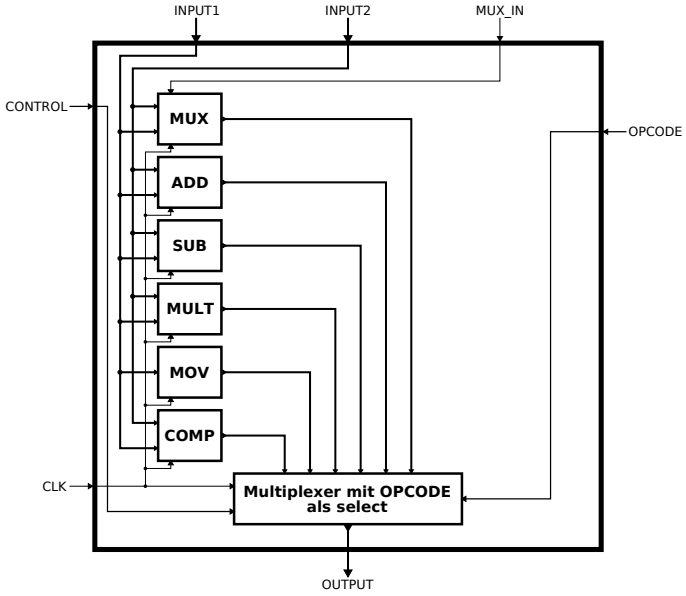


Abbildung 4.9: Funktionseinheit der RT-Ebene

den Operanden um Festkommazahlen mit zehn Bit für die Nachkommastellen handelt. Die beiden Operanden (egal ob 18 oder 32 Bit) werden miteinander multipliziert, das Ergebnis um zehn Bit nach rechts verschoben und anschließend auf 32 Bit gekürzt bzw.

Zur Aufbewahrung der Konfigurationsdaten für die SubVHM dient ein einziger Speicher, welcher sowohl die Daten für die Kommunikation als auch die Operationscodes für die Funktionseinheit zu Verfügung stellt. Aufgrund dieser Tatsache ist die Konfiguration wesentlich einfacher als die der Logikebene. Sie wird ebenfalls von außen gesteuert und verwendet hierfür folgende Ein- bzw. Ausgangssignale (vgl. Abb. 4.10)

- *conf_pos* (4 Bit Eingang): spezifiziert die Speicheradresse, welche beschrieben werden soll
- *conf_data* (15 Bit Eingang): stellt den Eingang dar, an welchem die Konfigurationsdaten anliegen
- *conf_req* (1 Bit Eingang): startet die Übernahme der Konfigurationsdaten
- *conf_ack* (1 Bit Ausgang): quittiert die Übernahme der Konfigurationsdaten.

Die Reihenfolge der Schritte für die Konfiguration ist ebenfalls gleich der der Logikebene. Als erstes werden die Konfigurationsdaten an das Signal *conf_data* angelegt. Der an *conf_pos* angelegte Wert gibt die zu beschreibende Speicheradresse des Konfigurationsspeichers an. Das Eingangssignal *conf_req* ist direkt mit dem *write_enable*-Eingang des Konfigurationsspeichers und dem *conf_ack*-Ausgang verbunden. Wird dieses nun auf 1 gesetzt, so werden einerseits die Konfigurationsdaten in die angegebene Speicheradresse geschrieben, andererseits wird die Übernahme nach außen signalisiert, indem *conf_ack* durch *conf_req* auf 1 gesetzt wird. Im letzten Schritt wird *conf_req* von außen wieder auf 0 zurückgesetzt, woraufhin der *conf_ack*-Ausgang der SubVHM

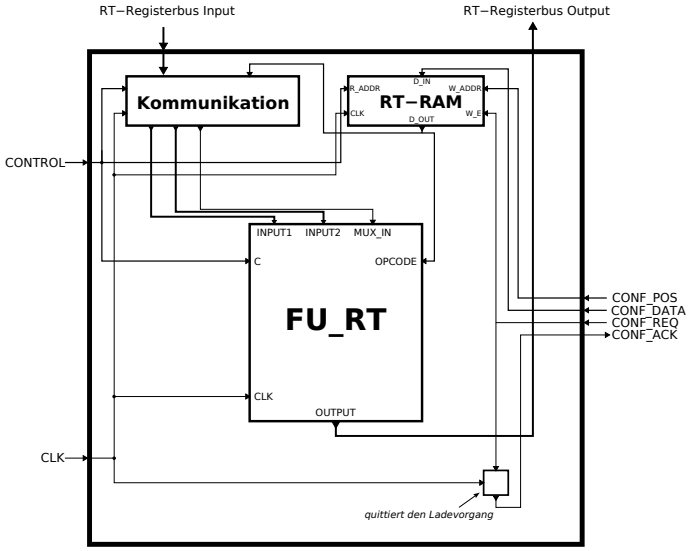


Abbildung 4.10: Die SubVHM der RT-Ebene

Abbildung 4.11: Instruktionswort der RT-Ebene

ebenfalls den Wert 0 annimmt. Der genaue Anschluss der externen Signale kann in Abbildung 4.10 noch einmal näher betrachtet werden.

4.1.3 VHM

Die VHM vereint und steuert alle vorher beschriebenen Komponenten. Sie beinhaltet eine oder mehrere SubVHMs, ist für die Behandlung der Konfigurationsanfragen verantwortlich, steuert den Programmablauf und kümmert sich um das Einlesen der Eingabewerte und die Ausgabe der Ergebnisse. Weitere in ihr enthaltene Komponenten sind die Registerfiles für die RT- und die Logikebene, sowie ein Registerfile für die aus dem VHBC übernommenen Konstanten. Die in dieser Diplomarbeit implementierte VHM besitzt vier Logik-SubVHMs, 16 RT-SubVHMs und ein Konstanten-Registerfile für sieben 16-Bit Werte.

Die Steuerung des Programmablaufs ist ein recht einfacher Prozess. Sowohl für die RT-, als auch für die Logikebene existiert ein Zähler[8], welcher den momentanen Microzyklus adressiert. Diese Zähler laufen ständig[9] von Null bis zum Wert der Länge des kritischen Pfades der jeweiligen Ebene und sind direkt mit den Adresseingängen der ihrer Ebene entsprechenden Speicher verbunden. Die Längen der kritischen Pfade werden der VHM bei der Konfiguration mit übergeben. Aufgrund der Trennung der Kommunikation von den Funktionseinheiten und der Verwendung von Block-RAMs musste der Microzyklus in drei Phasen unterteilt werden (siehe Abb. 4.1 auf Seite 42). In

[8] In der momentanen Implementierung der VHM ist die maximale Pfadtiefe auf 16 Mikrozyklen festgelegt, da es für die meisten Anwendungen ausreichend ist. Es spricht aber nichts dagegen, diesen Wert größer oder kleiner zu gestalten.

[9] Nach jedem Mikrozyklus werden die Zähler um eins erhöht.

der ersten Phase wird die Adresse des momentanen Microzyklus gesetzt, um dann in der zweiten Phase die Daten der Speicher für die Kommunikation verwenden zu können. In der dritten Phase wird dann die Berechnung durchgeführt und die Zähler für die Microzyklen inkrementiert. Jede dieser Phasen dauert einen Systemtakt, so dass ein kompletter Microzyklus drei Systemtakte in Anspruch nimmt. Alle Komponenten der VHM werden auf diese Art gesteuert und synchronisiert, da jede direkt mit den Zählern für die Mikrozyklen und dem Steuersignal für die Phase verbunden ist. Dabei gibt es noch zwei spezielle Fälle. Zum einen den, im welchem die externen Eingabewerte übernommen werden, wenn der Microzyklenzähler einer Ebene Null ist und zum anderen den, in dem die Ergebnisse in die Ausgaberegister der VHM geschrieben werden, wenn der Microzyklenzähler einer Ebene gleich dem Wert der Länge des entsprechenden kritischen Pfades ist. Da in VHDL nur immer ein einziger Prozeß ein Signal beschreiben darf, dennoch aber externe Werte übernommen werden sollen, mussten für jede Ebene zwei Register Files implementiert werden. Das eine dient den einzelnen Funktionseinheiten als Ausgaberegister. Dieses gesamte Register File wird am Ende eines jeden Mikrozyklus komplett in das andere Register File geschrieben, welches den SubVHMs als Wertebasis für die Kommunikation dient, falls sie Ergebnise anderer SubVHMs benötigen. Die Übernahme des externen Werte funktioniert dann so, dass im ersten Mikrozyklus nicht das Register File mit den Ergebnisen, sondern die externen Werte in das andere geschrieben werden. Es wird dabei nur die Anzahl von Eingabewerten übernommen, welche im Header des VHBC (Abb. 2.15 auf Seite 30) angegeben ist. Der Rest wird mit den Werten aus dem ersten Registerfile gefüllt. Die Ausgabe der Werte der gesamten Berechnung erfolgt, wenn der Zähler für die Mikrozyklen gleich dem Wert für die Länge des kitischen Pfades der entspechenden Ebene ist. Abbildung 4.12 verdeutlicht diesen Ablauf noch einmal grafisch.

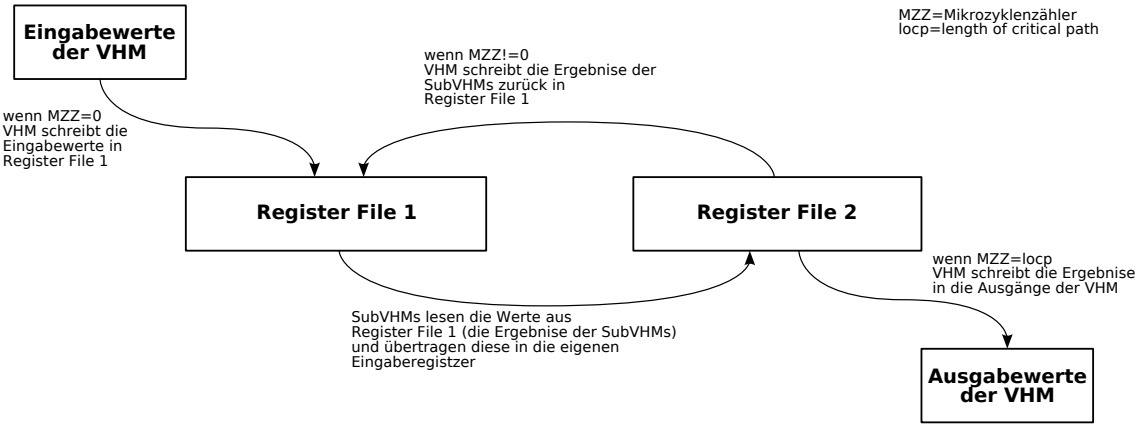


Abbildung 4.12: Zusammenspiel der beiden Register Files, Übernahme externer Werte und die Ausgabe der Berechnungsergebnise.

Um die Komplexität des Konfigurationsvorganges zu verringern, wurde auf einen hardwarseitigen Decoder zum Einlesen der VHBC-Daten verzichtet. Die Decodierung der VHBC-Daten wird nun dem System überlassen, welches sich der VHM als Rechenressource bedient. Das Ergebnis ist eine einfache Konfigurationsschnittstelle, welche durch verschiedene Ein- und Ausgabewerte gesteuert wird. Bei diesen Ein- uns Ausgabewerten handelt es sich um folgende (siehe auch Abb. 4.2):

- *conf* (1 Bit Eingang): entscheidet, ob sich die VHM im Konfigurations- oder Berechnungsmodus befindet
- *conf_pos* (11 Bit Eingang): spezifiziert die Zielkomponente und die Speicheradresse, welche beschrieben werden soll
- *conf_data* (64 Bit Eingang): stellt den Eingang dar, an welchem die Konfigurationsdaten anliegen
- *conf_req* (1 Bit Eingang): startet die Übernahme der Konfigurationsdaten
- *conf_ack* (1 Bit Ausgang): quittiert die Übernahme der Konfigurationsdaten.

Der erste Schritt für die Konfiguration der VHM ist das Setzen des Einganges *conf* auf 1, welches die VHM veranlasst vom Berechnungs- in den Konfigurationsmodus zu wechseln und sensitiv auf den *conf_req*-Eingang zu werden. Darauf erfolgt ein Reset der VHM mit Hilfe des *reset*-Eingangs, um die Mikrozyklenzähler der einzelnene Ebenen und des Phasenzählers auf Null zu setzen. Anschließend werden die Konfigurationsdaten am Eingang *conf_data* angelegt, welcher mit allen Komponenten der VHM verbunden ist. Der Eingang *conf_pos* spezifiziert dabei die Zielkomponente. Auf welche Art ist in Tabelle 4.3 zu sehen. Wird nun *conf_req* von außen auf 1 gesetzt, so leitet die VHM dieses an die in *conf_pos* angegebene Komponente weiter. Wie schon in 4.1.1 und 4.1.2 beschrieben, übernehmen die SubVHMs der einzelnen Ebenen die Konfigurationsdaten und quittieren das durch das Anlegen einer 1 an ihrem *conf_ack*-Ausgang. Dabei leitet die VHM das *conf_ack*-Signal der in *conf_pos* adressierten Komponente nach außen weiter. Im letzten Schritt der Konfiguration wird *conf_req* wieder auf 0 gesetzt, was die in *conf_pos* angegebene Komponente dazu veranlaßt, *conf_ack* ebenfalls auf 0 zu setzen und somit den Konfigurationsschritt beendet. Bei der Konfiguration der VHM gibt es noch zwei Spezialfälle. Zum einen die Übernahme der im VHBC enthaltenen Konstanten, wenn $conf\_pos(0\,to\,2) = \text{"}111\text{"}$ und $(conf\_pos(3) \vee conf\_pos(4)) =' 1'$ (3=die ersten vier Konstanten, 4=die letzten drei Konstanten). Zum anderen die Übernahme der im VHBC enthaltenen Werte für die Länge der kritischen Pfade und der Anzahl der berechnungsarelevanten Eingänge die beiden Ebenen, sowie die Anzahl der Konstanten für die RT-Ebene, wenn $conf\_pos(0\,to\,4) = \text{"}11100\text{"}$.

Im Vergleich zum Decoder hat sich somit geändert, dass die VHM das VHBC-Image nicht mehr selber dekodieren muss, sondern diese Aufgabe an ein übergeordnetes System abgibt, welches die VHM verwendet. Dadurch spart die VHM Ressourcen und gewinnt an Flexibilität. So ist es jetzt möglich, gezielt einzelne Teile der VHM umzukonfigurieren. Hat man z.B. ein Filter für ein Bild, welches auf eine 3x3 Umgebung eines Pixels angewandt wird und möchte dieses durch ein artgleiches Filter ersetzen, so ist lediglich das Austauschen der Filtermatrix notwendig, was durch das Ändern der Konstanten bewirkt wird, welches maximal zwei Konfigurationsschritte dauert.

Mit Hilfe welcher Mittel man auf die VHM zugreift ist nicht festgelegt. In dieser Diplomarbeit wurde für diesen Zweck das EDK[10] von Xilinx [13] verwendet. Es bietet die Möglichkeit die

[10] Embedded Development Kit

Tabelle 4.3: Belegung der Einganges *conf_pos*

conf_pos−Eingang (11Bit)

Bitposition	**0**	**1 2 3 4**		**5 6**	**7 8 9 10**
RT−Ebene	0	Nummer der zu konfigurierenden SubVHM		frei	Speicheradresse, welche beschrieben werden soll
Bitposition	**0**	**1 2**	**34**	**5 6**	**7 8 9 10**
Logikebene	1	Wert für den conf_ram−Eingang, welcher den zu konfigurierenden Teil einer SubVHM angibt	Nummer der zu konfigurierenden SubVHM	Nummer der zu konfigurierenden Funktionseinheit	Speicheradresse, welche beschrieben werden soll

Bitposition	**0 1 2 3 4**
Die Konstanten 0 bis 3 übernehmen	11101
Die Konstanten 4 bis 6 übernehmen	11110
Die Werte locp_(ll/rt), input_(ll/rt)_count und const_count übernehmen	11100

VHM an den 64-Bit breiten PLB[11] anzubinden. Der Zugriff erfolgt über memory-mapped Register, d.h. die VHM hat innerhalb des Systems eine Startadresse, welche auch die Adresse des ersten Registers der VHM ist und alle acht Byte folgt das nächste Register. Die im EDK implementierte VHM verfügt über zwölf 64-Bit Register, welche mit den Ein- und Ausgängen der VHM verbunden sind und ihre Verwendung und Steuerung ermöglichen.

[11] Processor Local Bus

4.2 VHBC-Compiler

Der VHBC-Compiler ist vollständig in Java implementiert, was die Lauffähigkeit auf nahezu allen Systemen gewährleistet. Ursprünglich verarbeitete der VHBC-Compiler auf der SimPrim-Bibliothek basierenden VHDL-Code und eine assemblerähnliche Sprache, welche von S. Lange entwickelt wurde [1]. Da die Implementierung eines vollwertigen VHDL-Compilers aber weit außerhalb des Rahmens dieser Diplomarbeit liegen würde, wurde sich dafür entschieden, den Compiler um das EDIF-Format (siehe 2.1) zu erweitern. Weitere Veränderungen, welche am VHBC-Compiler vorgenommen wurden sind die Implementierung einer Netzlistenoptimierung, die Möglichkeit der Verarbeitung von Instruktionen der RT-Ebene und eine völlig neue Generierung des VHBC auf der Grundlage der in Abschnitt 4.1 beschriebenen neuen VHM. Abbildung 4.13 zeigt den schematischen Arbeitsablauf des VHBC-Compilers und seine Unterteilung in die drei Hauptphasen: Eingabe, Code-Manipulation und Ausgabe, auf welche im Folgenden näher eingegangen wird.

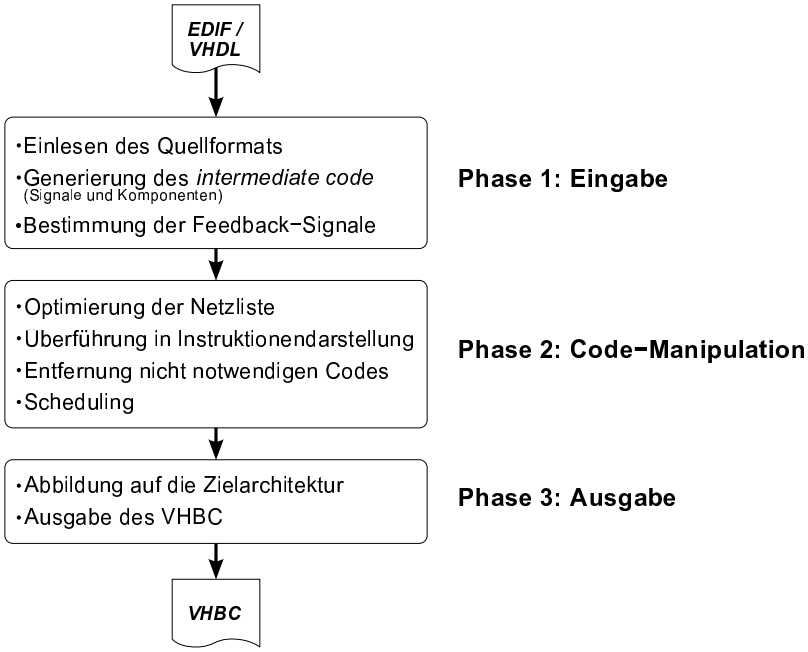


Abbildung 4.13: Arbeitsablauf des VHBC-Compilers

4.2.1 Phase 1: Eingabe

Die Eingabe ist die Phase im Arbeitsablauf des VHBC-Compilers, in welcher das Quellformat eingelesen und in eine, für den Rest des Compilers verständliche Zwischendarstellung überführt wird. Sie verbirgt alle spezifischen Einzelheiten des Quellformats vor der weiteren Verarbeitung. Dadurch ist bei der Aufnahme weiterer Quellformate lediglich eine Überführung dieser in die Zwischendarstellung notwendig, um mit ihnen arbeiten zu können. Bei der Zwischendarstellung handelt es sich um eine Menge von Komponenten, welche in der im Quellformat beschriebenen Schaltung enthalten sind, und den sie verbindenden Signalen. Da in dieser Arbeit ausschließlich

mit EDIF als Quellformat gearbeitet wurde, soll hier nicht näher auf VHDL als Quellformat eingegangen werden.

Um das Rad nicht ein weiteres Mal erfinden zu müssen, wurde zum Einlesen der EDIF-Netzlisten auf den EDIF-Parser des JHDL-Projekts [19] zurückgegriffen. JHDL ist eine Sammlung von Open Source FPGA CAD Werkzeugen, welche vom Configurable Computing Laboratory [20] der Brigham Young University [21] entwickelt werden und dem Benutzer die Möglichkeit geben, die Struktur und das Layout von Schaltungen zu entwickeln und diese mit Hilfe von Simulationen zu testen und zu debuggen.

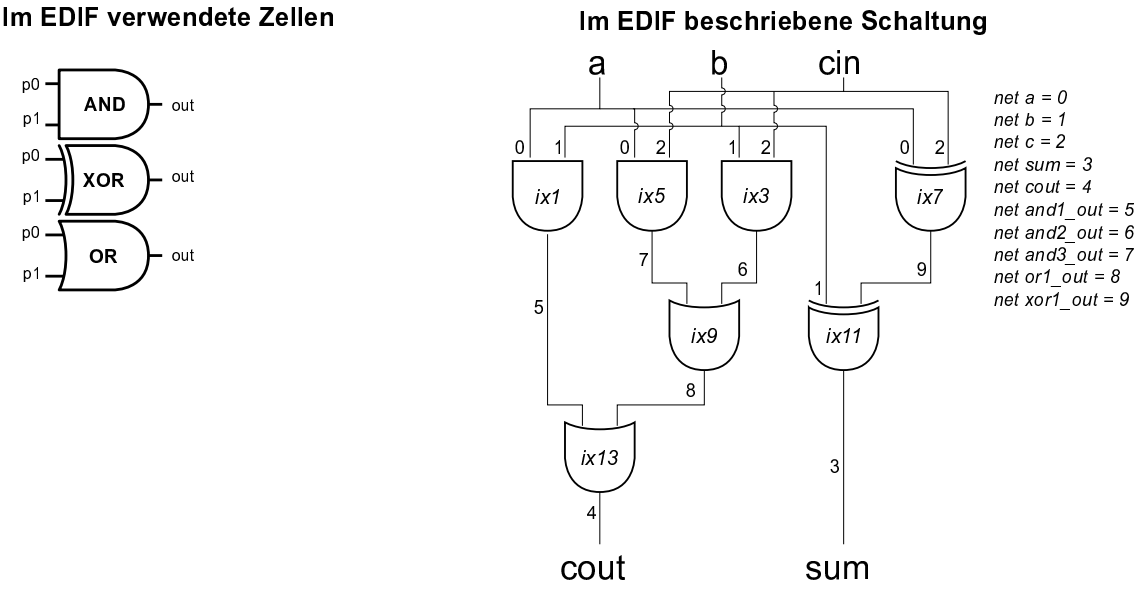
Der für diese Arbeit relevante Teil des JHDL-Projekts ist der EDIF 2 0 0 Parser. Da auch dieser in Java implementiert ist, gab es keine Probleme ihn in den VHBC-Compiler zu integrieren. Der EDIF-Parser des JHDL-Projekts bietet die Möglichkeit, auf jede einzelne Bibliothek und Zelle des Designs zuzugreifen. Dabei werden die im EDIF beschriebenen Zellen solange zerlegt, bis man bei den atomaren, nicht weiter zerlegbaren Zellen angekommen ist. Der EDIF-Parser des VHBC-Compilers wiederum verwendet das Wissen über die einzelnen Zellen, um aus ihnen Komponenten für die weitere Verarbeitung innerhalb des VHBC-Compilers zu erzeugen, welche alle für die spätere Verarbeitung notwendigen Informationen über die Eigenschaften der Zellen enthalten. Es handelt sich dabei zum einen um die Operation, welche die Zelle darstellt und zum anderen um die Verbindungen, welche die Zelle zu anderen Zellen besitzt. Besonders wichtig sind hierbei die Verbindungen, welche die Ein- und Ausgänge der Zellen miteinander verknüpfen. Jede dieser Verbindungen bekommt eine Signalnummer zugewiesen, mit deren Hilfe die beschriebene Schaltung eindeutig festgelegt ist. Es werden nur diejenigen Zellen in Komponenten der Zwischendarstellung überführt, welche atomare Zellen des EDIF repräsentieren. In Abbildung 4.14 ist das am Beispiele eines Volladdierers noch einmal näher dargestellt. Auf die Abbildung des EDIF wurde hierbei verzichtet, da es in einer lesbaren Größe zu viel Platz in Anspruch genommen hätte. Es befindet sich auf der beiliegenden CD.

Der letzte Arbeitsschritt des EDIF-Parsers bestimmt die Feedback-Signale. Ein Feedback-Signal stellt dabei ein Signal dar, welches als Eingang für eine Komponente dient, noch bevor es innerhalb eines Macrozyklus mit einem Wert beschrieben wurde. Es ermöglicht der VHM Werte von einem Macrozyklus zum nächsten zu übergeben, um mit ihnen weiterzurechnen. Das Resultat des EDIF-Parsers des VHBC-Compilers ist ein Vektor von Komponenten, eine Liste mit den dazugehörigen Signalen und der Information, welches der Signale ein Feedback darstellt.

4.2.2 Phase 2: Code-Manipulation

Die zweite Phase des VHBC-Compilers dient der Manipulation der eingelesenen Zwischendarstellung. Dabei handelt es sich um eine Netzlistenoptimierung, die Überführung in die Instruktionendarstellung, Scheduling und die Entfernung nicht notwendigen Codes. Es werden keine Veränderungen am Code durchgeführt, welche auf die spätere Zielarchitektur hinzielen.

Die Netzlistenoptimierung beschränkt sich momentan auf die Logikebene. Die Optimierung basiert auf normaler Logikoptimierung und ist in der Lage, logische Schaltungen merklich zu verkleinern. Das basiert auf der Tatsache, dass die meisten Synthesetools auf eine bestimmte Zie-



Komponentenname	Komponententyp	Eingangssignale	Ausgangssignale
ix1	AND	0,1	5
ix3	AND	1,2	6
ix5	AND	0,2	7
ix7	XOR	0,2	9
ix9	OR	7,6	8
ix11	XOR	1,9	3
ix13	OR	5,8	4

Abbildung 4.14: EDIF-Repräsentation eines Volladdierers und seine Darstellung in VHBCC-Komponenten

larchitektur hin optimieren und aus diesem Grund eine gewisse Konstellation von Komponenten favorisieren, während sie andere Komponenten aus Gründen der Nichtverfügbarkeit auf diesen Architekturen komplett weglassen. In Tabelle 4.4 sind noch einmal alle bedeutenden Optimierungen mit Erklärungen aufgeführt.

Anschließend wird die optimierte Netzliste in eine Liste von Instruktionen überführt, welche die Instruktionen des späteren VHBC darstellen. Hierbei werden die Komponenten der Zwischendarstellung in Instruktionen und die Signale in Register überführt. Eine Optimierung von Algorithmen auf RT-Ebene wäre über eine Schnittstelle zu einem CAS[12] denkbar, obgleich der Aufwand für die Implementierung den Nutzen meiner Meinung nach nicht abdecken würde, da man unter anderem erst die optimierungsfähigen Teile des Algorithmus extrahieren müsste.

Ein ASAP[13]-Scheduler bringt die Liste von Instruktionen in die für die spätere Ausführung richtige Reihenfolge. Hierbei werden die Instruktionen in Instruktionsebenen zusammengefasst. Kennzeichnend für eine Instruktionsebene ist, dass alle Instruktionen innerhalb einer Instruktionsebene

[12] Computer Algebra System
[13] **A**s **S**oon **A**s **P**ossible

Tabelle 4.4: Im VHBC implementierte Logikoptimierungen

Funktionsname	Erklärung
AOO_OAA(...)	$(A \wedge B) \vee (A \wedge C) \Rightarrow A \wedge (B \vee C)$ $(A \vee B) \wedge (A \vee C) \Rightarrow A \vee (B \wedge C)$
EQ_XOR(...)	$(\neg A \vee B) \wedge (A \vee \neg B) \Rightarrow A \equiv B$ $(\neg A \wedge B) \vee (A \wedge \neg B) \Rightarrow A \oplus B$ $(\neg A \wedge \neg B) \vee (A \wedge B) \Rightarrow A \equiv B$ $(\neg A \vee \neg B) \wedge (A \vee B) \Rightarrow A \oplus B$
EQ_XOR_2(...)	$(A \wedge B) \vee \neg(A \vee B) \Rightarrow A \equiv B$ $(A \vee B) \wedge \neg(A \wedge B) \Rightarrow A \oplus B$
INV(...)	verbindet, falls möglich, die Negation mit der vorhergehenden Operation (entfernt somit auch doppelte Negationen)
NOR_NAND(...)	$\neg(\neg(A\, op1\, B)\, op2\, \neg(A\, op3\, B)) \Rightarrow$ $(A\, op1\, B)\, op4\, (A\, op3\, B) \mid op1, op2, op3 \in$ $\{nand, nor\};\ op4 = \neg op2$
OR_NOR_AND_NAND(...)	$(\neg A \wedge \neg B) \Rightarrow \neg(A \vee B)$ $(\neg A \vee \neg B) \Rightarrow \neg(A \wedge B)$ $\neg(\neg A \wedge \neg B) \Rightarrow (A \vee B)$ $\neg(\neg A \vee \neg B) \Rightarrow (A \wedge B)$
removeRedundantComponents(...)	fasst Komponenten, welche die gleiche Operation auf die selben Signale anwendet zu einer Komponente zusammen

parallel und innerhalb eines Mikrozyklus (siehe Abb. 4.1 auf Seite 42) ausgeführt werden und somit nicht voneinander abhängig sind. D.h., dass keine Instruktion einer Instruktionsebene ein Ergebnis einer anderen Instruktion seiner oder einer vorangegangenen Ebene benötigt und alle Eingänge der Instruktionen einer Ebene durch Instruktionen der darüber liegenden Ebenen zu Verfügung gestellt werden bzw. auf externe Eingänge verweisen. In Abbildung 4.14 ist gut zu sehen, wie die Instruktionen (*ix1*, *ix3*, *ix5*, *ix7*), (*ix9*, *ix11*) und (*ix13*) jeweils eine Instruktionsebene bilden. Eine Ausnahme bei den Abhängigkeiten von den Ein- und Ausgängen der Instruktionen stellen die Feedback-Register dar. Würde man Feedback-Register nicht erkennen und nicht gesondert behandeln, so hätte man bei den Abhängigkeiten der Instruktionen untereinander einen Kreisschluss, da z.B. eine Instruktion mit seinem eigenen Ergebnis im nächsten Makrozyklus weiter rechnen kann, aber aufgrund dieser Abhängigkeit nicht mit sich selbst in einer Instruktionsebene stehen dürfte, was diese Instruktion vollkommen unbrauchbar machen würde.

Der momentan implementierte ASAP-Scheduler ist nur eine mögliche Implementierung. So besteht die Möglichkeit eigene Scheduler zu realisieren, welche die Liste von Instruktionen in eine, für die eigenen Bedürfnisse entsprechende Form bringen, um beispielsweise eine Datenflussarchitektur zu verwirklichen (Abb. 4.15).

Um nicht notwendigen Code zu entfernen, erfolgt nach der Netzlistenoptimierung eine Konsistenzprüfung der eingelesenen Schaltung, indem alle externen Signale, Feedback-Register und Flip-Flop-Ausgänge der Schaltung in einer Liste möglicher Eingänge zusammengefasst werden.

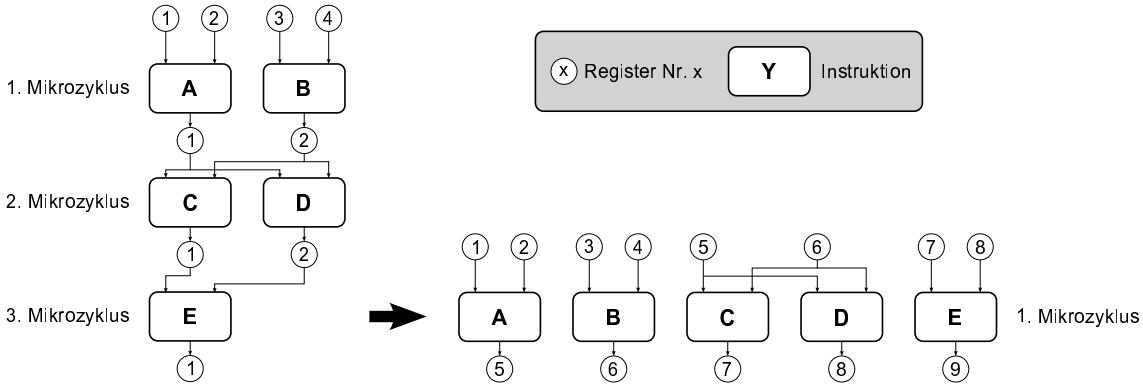


Abbildung 4.15: Umwandlung einer Schaltung in eine Darstellung, welche der VHM ein Datenflussverhalten ermöglicht und in welchem nach nur einem Microzyklus ein Ergebnis ausgegeben werden kann. Diese Transformation könnte man mit Hilfe eines neuen Schedulers realisieren.

Anschließend sucht man nach einer Komponente, deren Eingänge in dieser Liste vorhanden sind. Darauf hin wird diese Komponente aus der Liste der vorhandenen Komponenten entfernt und in eine Resultatsliste eingetragen. Ihre Ausgänge werden zu der Liste von möglichen Eingangssignalen hinzugefügt. Diese Suche wird so lange wiederholt, bis keine weitere Komponente gefunden werden kann, welche in die Resultatsliste eingefügt werden kann. Es bleiben die Komponenten übrig, deren Eingänge nicht vollständig vorhanden sind, dadurch keine brauchbaren Ergebnise liefern und somit auch keine Relevanz für die Schaltung besitzen.

4.2.3 Phase 3: Ausgabe

Die letzte Phase in der Codeverarbeitung ist die Ausgabe. Sie bildet den in Phase 1+2 bearbeiteten Code auf die Zielarchitektur ab und generiert den VHBC. Die Schwierigkeit dabei besteht darin, dass die bis hierher erstellte Liste von Instruktionsebenen ’horizontal’ zerteilt und auf die SubVHMs verteilt werden muss. Dabei müssen die hardwareseitigen Beschränkungen der SubVHMs beachtet werden. Das bedeutet, dass die Liste von Instruktionsebenen in Blöcke zerlegt wird, wobei jeder Block den Code für eine SubVHM darstellt und ihren Vorgaben gerecht werden muss. Diese Beschränkungen beziehen sich zum einen auf die Tiefe und Breite eines Blocks und zum anderen auf die Anzahl der Ein- und Ausgänge. So darf auf Logikebene ein Block nicht mehr als 16 Instruktionsebenen und eine Instruktionsebene nicht mehr als 16 Instruktionen beinhalten. Weiterhin dürfen pro Ebene nicht mehr als 16 Eingänge verwendet werden. Die Schwierigkeit bei der Zerlegung liegt darin, dass man bereits berechnete Wert berücksichtigen muss, welche im späteren Verlauf der Berechnung noch benötigt werden und somit unter Umständen über mehrere Instruktionsebenen gültig bleiben müssen. Es besteht entweder die Möglichkeit, die Position der Instruktion, welche den Wert berechnet hat, bis zu der Ebene vor der Ebene, in welcher der Wert das letzte Mal verwendet wird, mit leeren Instruktionen zu füllen, was einem Weiterreichen des Ergebnisses an dieser Position entspricht. Oder man reicht das Ergebnis mit Hilfe von MOV-Instruktionen weiter, was mehr Flexibilität innerhalb des VHBC ermöglicht. Auf RT-Ebene gestaltet sich die Aufteilung der Instruktionen auf verschiedene Blöcke wesentlich einfacher, da

momentan nur eine Instruktion pro Instruktionsebene vorhanden ist. Die maximale Tiefe einer RT-SubVHM beträgt ebenfalls 16 Instruktionsebenen und sie besitzt lediglich die drei Eingänge der einzigen Instruktion[14] pro Ebene.

Zerlegung des Codes auf Logikebene

Die Zerlegung besteht aus zwei Teilen. Im ersten Teil werden für alle Register des Originalcodes die Gültigkeitsbereiche bestimmt. Hierbei wird zwischen den Gültigkeitsbereichen als Eingang und denen als Ausgang unterschieden. Ein Eingangsbereich eines Registers ist der Bereich von der ersten Instruktionsebene, in der das Register das erste Mal als Eingang einer Instruktion dient, bis zur letzten Instruktionsebene, in der das Register das letzte Mal als Eingang fungiert. Der Ausgangsbereich ist der Bereich, in dem ein Register relevant ist (Ausgang einer Instruktion bis zur letzten Verwendung). Im zweiten Teil geschieht die eigentliche Zerlegung des Codes. Zuerst werden die Flip-Flops, d.h. alle zu einem Flip-Flop gehörenden Instruktionen aus dem Code extrahiert und in neue Codeblöcke eingepasst. Dabei wird darauf geachtet, dass alle Instruktionen die für die Realisierung eines Flip-Flop notwendig sind, in einem Codeblock untergebracht werden. Falls kein aufnahmefähiger Codeblock mehr vorhanden ist, so wird ein neuer erzeugt. Anschließend wird der Rest des Originalcodes in Codeblöcke zerteilt, indem solange vom verbleibenden Code Codeblöcke abgespalten werden, bis keine weitere Zerteilung möglich ist. Die Bildung eines neuen Codeblocks geschieht im Detail so, dass alle Instruktionsebenen von oben nach unten abgearbeitet werden. Hierbei wird zuerst die Liste der vom neuen Codeblock verwendeten Eingangsregister mit Hilfe der Gültigkeitsbereiche aktualisiert, indem diejenigen Register aus der Liste der Eingangsregister entfernt werden, deren Eingangsgültigkeitsbereich vor der momentanten Instruktionsebene endet. Danach wird jede Instruktion der Instruktionsebene des restlichen Originalcodes daraufhin geprüft, ob sie in die entsprechende Instuktionsebene des neuen Codeblocks passt. Zuerst werden hierfür die zwei Eingänge der einzufügenden Instruktion als weitere Eingänge der momentanen Instruktionsebene registriert und überprüft, ob in der momentanen und allen vorhergehenden Instruktionsebenen nicht mehr als 16 Eingänge verwendet wurden. Im Anschluss wird der neue Codeblock nach belegbaren MOV-Instruktionen durchsucht, welche das Ergebnis der einzufügenden Instruktion bis zum Ende seines Gültigkeitsbereiches weiterleiten. Sollten nicht ausreichend freie MOV-Instruktionen vorhanden sein, so werden diese in den entsprechenden Instruktionsebenen eingefügt. Als letztes wird die aktuelle Instruktion in den neuen Codeblock eingefügt. Sollten an einer Stelle dieses Einfügungsprozesses die Beschränkungen eines Codeblocks nicht eingehalten werden können (max. 16 Instruktionen und 16 Eingänge pro Instruktionsebene), so werden alle Änderungen rückgängig gemacht und mit der nächsten Instruktion fortgeschritten. Dieser Prozess wird so lange fortgesetzt, bis alle Instruktionen des Originalcodes auf neue Codeblöcke aufgeteilt sind.

[14] Zwei reguläre Eingange, an denen Werte anliegen, mit denen gerechnet wird und einen 'Select'-Eingang für den in der FU implementierten Multiplexer

Zerlegung des Codes auf RT-Ebene

Die Zerlegung des Originalcodes in Codeblöcke gestaltet sich für die RT-Ebene aufgrund der simpleren Struktur der Blöcke wesentlich einfacher. Der erste Schritt ist gleich dem der Logikebene und dient der Bestimmung der Gültigkeitsbereiche der verwendeten Register. Im zweiten Schritt werden für alle Instruktionen, welche ein Feedbackregister beschreiben, eigene Codeblöcke erstellt, um sicherzugehen, dass die Feedbackregister nicht von anderen Instruktionen überschrieben werden. Der dritte und letzte Schritt dient der Zerlegung des restlichen Codes. Hierfür wird der Originalcode solange durchlaufen, bis alle Instruktionsebenen des Originalcodes keine Instruktionen mehr enthalten. Die Erzeugung eines neuen Codeblocks geschieht so, dass einer Instruktionsebene eine Instruktion entnommen und diese in die entsprechende Ebene des neuen Codeblocks eingefügt wird. Anschließend wird in jede Instruktionsebene des neuen Codeblocks, welche sich im Gültigkeitsbereich des Ausgangs der eingefügten Instruktion befinden, eine MOV-Instruktion eingefügt, um das Ergebnis der Instruktion bis zum Ende des Gültigkeitsbereiches weiterzureichen.

Ist die Ebenenliste in Blöcke zerlegt, werden die für die Kommunikation zwischen den SubVHMs notwendigen Informationen bestimmt. Das geht so vonstatten, dass jedes Register des Designs, welches weder einen externen Ein- noch Ausgang repräsentiert, überprüft wird, ob es außer im eigenen Block, in dem es den Ausgang einer Instruktion repräsentiert, noch in anderen Blöcken als Eingang fungiert. Im Ergebnis erhält der Compiler dadurch Informationen darüber, ob für einen der Eingänge der SubVHM eine Kommunikation notwendig ist und wenn, mit welcher anderen SubVHM kommuniziert werden muss, sowie welches Register dieser SubVHM benötigt wird. Diese Informationen werden für jede Instruktionsebene jeder SubVHM gesammelt, um aus ihnen die Kommunikationsanweisungen zu erstellen.

Nachdem alle notwendigen Arbeiten zur Abbildung auf die Zielarchitektur abgeschlossen und alle erforderlichen Informationen zusammengetragen wurden, erfolgt die Ausgabe des VHBC. Wie schon in der ersten Version des VHBC, besteht dieser aus einem Header und einem Datenteil (Abb. 4.16). Der Header (Abb. 4.17) enthält, neben den VHBC als solchen kennzeichnenden Zeichen (der VHBC und somit der Header beginnt mit ’VHBC’) und der Versionsnummer, die für die Ausführung wichtigen Laufzeitinformationen:

- Länge des kritischen Pfades der Logik- und der RT-Ebene,
- Anzahl der verwendeten SubVHMs für die Logik- und für die RT-Ebene,
- Anzahl der verwendeten externen Ein- und Ausgänge für die Logik- und für die RT-Ebene,
- Anzahl der verwendeten Konstanten für die RT-Ebene

Der auf den Header folgende Datenteil beginnt mit der Anzahl im Header angegebener Konstanten. Anschließend werden die Daten für die SubVHMs geschrieben, wobei zuerst alle SubVHMs der Logikebene und danach alle SubVHMs der RT-Ebene behandelt werden. Der Datensatz einer SubVHM der Logikebene setzt sich aus den Kommunikationsdaten für alle Instruktionsebenen,

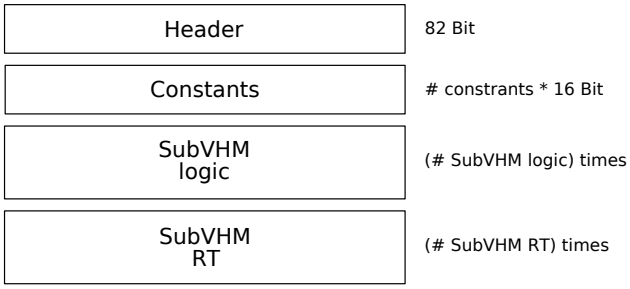


Abbildung 4.16: struktureller Aufbau des VHBC

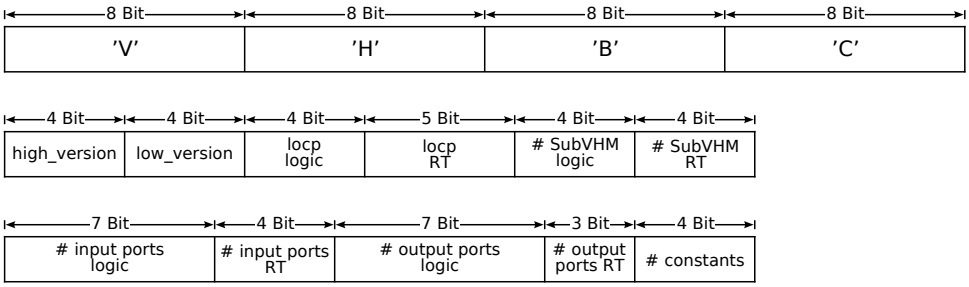


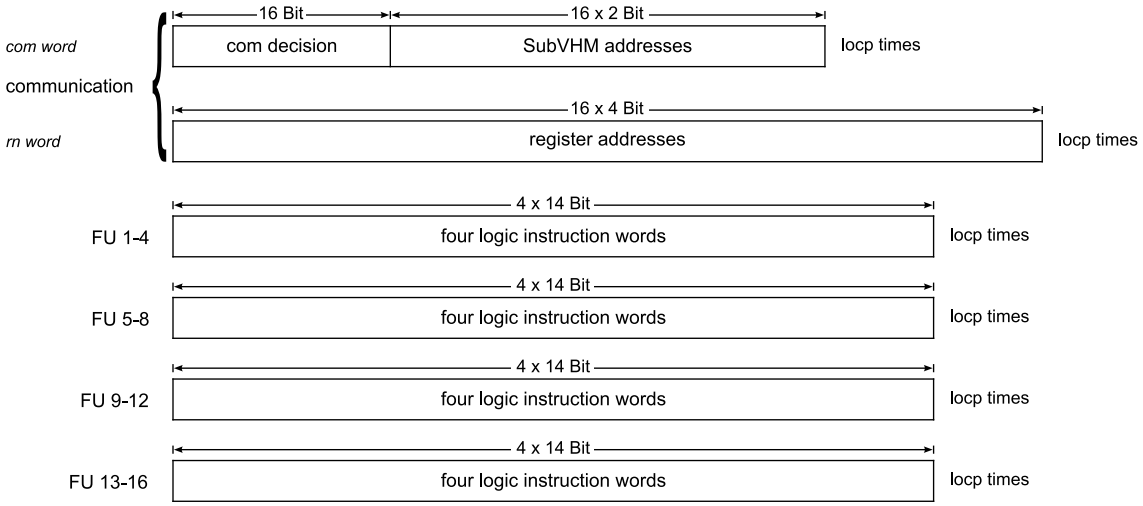
Abbildung 4.17: Aufbau des Header des VHBC (locp = length of critical path)

gefolgt von den Daten für die FUs zusammen. Dabei werden die FUs immer in Vierergruppen geschrieben, was seinen Ursprung in der gemeinsamen Nutzung eines BlockRAMs durch vier FUs hat. Von dieser Vierergruppe werden ebenfalls alle Instruktionsebenen geschrieben, bevor zur nächsten Vierergruppe übergegangen wird. Der Datensatz für eine SubVHM der RT-Ebene ist einfacher strukturiert. Es handelt sich hierbei um ein einziges Instruktionswort pro Instruktionsebene (Abb. 4.11 auf Seite 53), welches die Daten für sowohl die Kommunikation, als auch für die FU beinhaltet. Hinzugesagt werden muss, dass immer nur der Länge des kritischen Pfades entsprechend viele Instruktionsebenen und der Anzahl der SubVHMs entsprechend viele SubVHM-Datensätze geschrieben werden, was den VHBC kompakter hält.

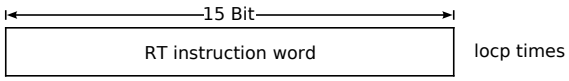
4.2.4 Anmerkung zu den Konstanten

Um für die Verwendung von Konstanten innerhalb einer Schaltung nicht einen neuen Instruktionstyp einführen zu müssen, wurde eine spezielle Eingangsbenennung eingeführt. Alle Eingänge eines Designs sind in einer EDIF-Netzliste innerhalb der 'interface'-Sektion der Hauptzelle Zelle beschrieben. Um nun einen Eingang als eine Konstante zu kennzeichnen, wurde die Benennung des Eingangs so festgelegt, dass er mit 'CONST_' beginnt und danach die gewünschte Konstante folgt. Soll z.B. ein Eingang konstant -5 sein, so wird dieser mit dem Namen 'CONST_-5' versehen. Der Compiler erkennt somit anhand des 'CONST_', dass es sich um eine Konstante handelt. Die Verwendung von gebrochenen Zahlen ist ebenfalls möglich. Der Compiler wird entweder auf der Grundlage der Standardeinstellung oder dem, über einen Parameter dem Programm übergebenen Wert, die gebrochene Zahl in eine Festkommazahl umwandeln.

Da auf Logikebene 64 Eingänge zur Verfügung stehen und es in ihr nur zwei annehmbare Werte



(a) Logikebene



(b) RT-Ebene

Abbildung 4.18: Aufbau der Datensegmente für die SubVHMs der Logikebene und der RT-Ebene (locp = length of critical path)

gibt, wurde in ihr auf die Einführung von Konstanten verzichtet. Auf RT-Ebene ergeben sich aufgrund der architektonischen Eigenschaften dieser einige Einschränkungen. Die RT-Ebene besitzt 16 32-Bit Register, von denen im ersten Schritt eines Makrozyklus maximal neun der Register mit externen Eingabewerten gefüllt werden können. Somit bleiben für die Verwendung von Konstanten noch sieben Register übrig, was der maximale Anzahl an Konstanten entspricht.

Kapitel 5

Ergebnisse

In diesem Kapitel werden einige Beispiele vorgestellt, um die Leistungsfähigkeit der VHM zu zeigen und mit der anderer Ansätze, wie dem *System Generator for DSP* von Xilinx, zu vergleichen. Dabei wird nicht ausschließlich auf die Verarbeitungsgeschwindigkeit geachtet, sondern es werden auch Flächenverbrauch, Konfigurationsaufwand und Flexibilität gegenübergestellt. Für alle Beispiele und Tests diente der Xilinx Virtex2P xc2vp30-7ff896. Zuerst erfolgt aber eine Auflistung der Synthesewerte der VHM, um sie mit den nachfolgenden Beispielen vergleichen zu können.

5.1 Ressourcenverbrauch und Geschwindigkeit der VHM

Die VHM ist, zumindest in Bezug auf den verwendeten FPGA, eine große und komplexe Maschine. Dies ist vor allem der Tatsache geschuldet, die VHM nicht für ein spezielles Problem konzipiert wurde, sondern ein möglichst breites Spektrum an Aufgaben bewältigen soll (Realisierung von Logikschaltungen, Berechnungen im DSP-Bereich).

Wie der in Abbildung 5.1 dargestellte Synthesereport der VHM zeigt, belegt diese 76 Prozent des FPGAs. Dieser Wert sinkt auf 72 Prozent bei der Verwendung der 18-Bit statt der 32-Bit Multiplikation. Dennoch bricht dadurch die maximale Taktrate nicht gänzlich ein und liegt mit 104 MHz (bzw. 170 MHz bei der Verwendung der 18-Bit Multiplikation) in einem vertretbaren Bereich. Leider liegt die effektive Taktrate der VHM nur bei einem Drittel der eben genannten 104 MHz, da ein Mikrozyklus der VHM insgesamt 3 Takte benötigt (Adresse an die BlockRAMs anlegen, Kommunikation, Berechnung). Dies führt zu einer effektiven Taktrate von rund 35 MHz.

5.2 Messungen

Das sich die VHM in Puncto Geschwindigkeit und Ressourcenverbrauch nicht mit einer problemspezifisch generierten Lösung wie denen des *System Generator* von Xilinx messen kann ist einleuchtend. Interessant wäre aber zu sehen, wie groß der Unterschied ist. Dazu werden im Folgenden zwei Beispiele betrachtet. Es handelt sich dabei um einen Medianfilter, welcher auf eine 3x3

```
=====================================================================
*                            Final Report                           *
=====================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                    10490  out of  13696    76%
 Number of Slice Flip Flops:           2730  out of  27392     9%
 Number of 4 input LUTs:              20253  out of  27392    73%
 Number of bonded IOBs:                 624  out of    556   112% (*)
 Number of BRAMs:                        64  out of    136    47%
 Number of MULT18X18s:                   64  out of    136    47%
 Number of GCLKs:                         1  out of     16     6%

WARNING:Xst:1336 -  (*) More than 100% of Device resources are used


=====================================================================
TIMING REPORT


Clock Information:
------------------
-----------------------------------+-------------------------+-------+
Clock Signal                       | Clock buffer(FF name)   | Load  |
-----------------------------------+-------------------------+-------+
clk                                | BUFGP                   | 2794  |
-----------------------------------+-------------------------+-------+

Timing Summary:
---------------
Speed Grade: -7

   Minimum period: 9.592ns (Maximum Frequency: 104.255MHz)
   Minimum input arrival time before clock: 4.686ns
   Maximum output required time after clock: 3.293ns
   Maximum combinational path delay: No path found

=====================================================================
```

Abbildung 5.1: Auszug aus dem Synthesereport der VHM.

Umgebung angewendet wird, und einen Farbraumkoverter. Diese wurden sowohl mit dem *System Generator* als auch mit der VHM getestet. Die Messwerte für den *System Generator* sind dem Synthesereport entnommen. Da die VHM bereits synthetisiert und damit ihre maximale Taktrate bereits bekannt ist, kann anhand der Ausgaben des VHBC-Compilers auf die Leistungsfähigkeit der VHM geschlossen werden.

Text

5.2.1 3x3 Medianfilter (Anwendung in der Bildverarbeitung)

Die Realisierung eines Medianfilters mit Hilfe des *System Generator* ist einfach. Per drag'n'drop werden alle benötigten Komponenten zusammengestellt und miteinander verbunden. Da der *System Generator* nur über Vergleichsoperationen verfügt, welche als Ergebnis einen boolschen Wert liefern, musste die Schaltung unter Verwendung von Multiplexern realisiert werden. Die anschließende Generierung des VHDL-Codes und dessen Synthese ergab folgende Werte.

```
================================================================================
*                            Final Report                                      *
================================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7


 Number of Slices:                      684  out of  13696     4%
 Number of Slice Flip Flops:            550  out of  27392     2%
 Number of 4 input LUTs:                656  out of  27392     2%
 Number of bonded IOBs:                  83  out of    556    14%
 Number of GCLKs:                         1  out of     16     6%


================================================================================
TIMING REPORT


Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 550   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -7

   Minimum period: 4.197ns (Maximum Frequency: 238.288MHz)
   Minimum input arrival time before clock: 3.782ns
   Maximum output required time after clock: 4.064ns
   Maximum combinational path delay: No path found

================================================================================
```

Geschwindigkeit

Wie zu sehen ist, kann der vom *System Generator* erzeugt Medianfilter mit 238 MHz getaktet werden. Gegenüber diesem Wert erscheint die VHM recht langsam. Die Ausgabe des VHBC-Compilers verrät, das der von mir implementierte Medianfilter zehn Mikrozyklen benötigt bis das Ergebnis ausgegeben wird. Das bedeutet, dass der mit Hilfe der VHM realisierte Medianfilter mit effektiv 3,5 MHz arbeitet.

5.2.2 Farbraumkonverter (RGB ->Y Pr Pb)

Bei diesem Beispiel handelt es sich um ein simples mathematische Problem. Die Konvertierung erfolgt nach den Formeln
$Y = 0,299 * R + 0,587 * G + 0,114 * B;\ Pr = 0,71327 * (R - Y);\ Pb = 0,56433 * (B - Y)$
Der *System Generator* optimiert die Formel für Y weiter zu $Y = 0,299 * (R - G) + (0,114 * (B - G) + G$ was eine Multiplikation einspart, die Berechnung aber um einen Schritt verlängert. Abbildung 5.2 zeigt einen Auszug aus dem Synthesereport der *System Generator* Implementierung.

Geschwindigkeit

Wie der Abbildung 5.2 zu entnehmen ist, ist die maximal zulässige Taktrate für die *System Generator* Implementierung des Farbkonverters 122 MHz. Der gleiche Konverter benötigt für die

```
=========================================================================
*                            Final Report                               *
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                      468  out of  13696     3%
 Number of Slice Flip Flops:            599  out of  27392     2%
 Number of 4 input LUTs:                799  out of  27392     2%
 Number of bonded IOBs:                  63  out of    556    11%
 Number of GCLKs:                         1  out of     16     6%

=========================================================================
TIMING REPORT


Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 639   |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -7

   Minimum period: 8.177ns (Maximum Frequency: 122.298MHz)
   Minimum input arrival time before clock: 1.418ns
   Maximum output required time after clock: 3.293ns
   Maximum combinational path delay: No path found

=========================================================================
```

Abbildung 5.2: Auszug aus dem Synthesereport des Farbkonverters.

VHM fünf Mikrozyklen, zwei für den Y-Wert, zwei für Pr und Pb und einen für die Ausgabe. Das bedeutet, dass die VHM effektiv mit rund 7 MHz arbeiten würde.

5.3 Flächenvergleich

Die VHM verbraucht fast 20-mal mehr Ressourcen als die *System Generator* Implementierung (VHM 76 Prozent, *SG* 3-4 Prozent). Hier zeigt sich wie speziell diese Implementierungen sind. Das ist vor allem darauf zurückzuführen, dass der *System Generator* nur notwendige Komponenten implementiert, während die VHM immer eine feste Anzahl von Funktionseinheiten beinhaltet, egal ob diese benötigt werden oder nicht. An diesem Sachverhalt wird sich aber auch nichts ändern, da das eine der grundlegenden Eigenschaften der VHM darstellt.

5.4 Konfigurationsaufwand

Bei der Betrachtung des Konfigurationsaufwands stellt sich sofort heraus, dass die VHM auf diesem Gebiet dem *System Generator* überlegen ist. Während die VHM mit Konfigurationsdaten in der Größe von ein paar Hundert Byte auskommt (der Medianfilter benötigt z.B. 180 Byte), muss im Fall des *System Generator* der komplette FPGA neu konfiguriert werden (rund 1,4 MB Konfigurationsdaten). Desweiteren ist eine gezielte Konfiguration ausgewählter Komponenten nur mit großen Aufwand möglich, wohingegen diese Eigenschaft einen zentralen Bestandteil des Konfi-

gurationsvorgangs der VHM darstellt.

5.5 Flexibilität

Bei der Flexibilität sehe ich beide Ansätze gleich auf. Jeder ist auf seine Art sehr anpassungsfähig. Die Implementierung mit Hilfe des *System Generator* gewinnt seine Flexibilität dadurch, dass die Ressourcen des FPGAs effektiver genutzt werden als bei der VHM und sich somit ein breiteres Anwendungsgebiet erschließt. Die VHM hingegen ist nicht auf rekonfigurierbare Hardware beschränkt und könnte auch in fester Hardware realisiert werden, ohne seine Rekonfigurierbarkeit einzubüßen. Weiterhin ist sie durch die wesentlich schnellere Konfiguration in der Lage schnell und unkompliziert zwischen verschiedenen Konfigurationen zu wechseln.

5.6 Fazit

Es lässt sich darüber streiten ob ein direkter Vergleich dieser beiden spezialisierten Ansätze sinnvoll ist. Auch wenn die VHM in einigen Bereichen weit hinter den Implementierungen des *System Generator* ist, so stellt sie doch eine rechenstarke flexible rekonfigurierbare Hardware dar.

Alle in diesem Kapitel angesprochenen Beispiele befinden sich bis auf den Farbkonverter auf der beiliegenden CD.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Was wurde erreicht?

Es wurde eine lauf- und synthesefähige VHDL-Beschreibung der VHM erstellt und der vorhandene VHBC-Compiler überarbeitet, erweitert und verbessert.

Die VHM ist in ihrer nun vorliegenden Version in der Lage, sowohl komplexe Berechnungen, als auch einfache logische Schaltungen zu realisieren. Ihr Anwendungsgebiet geht somit über das ursprünglich von Sebastian Lange angestrebte Ziel einer Maschine, welche die Funktionalität beliebiger Hardware übernehmen kann, hinaus, bis in den Bereich von digitalen Signalprozessoren. Die Erweiterung der VHM um die RT-Ebene ist hier besonders herauszuheben, da sie es ist, welche die VHM zu einer leistungsstarken und einsatzfähigen rekonfigurierbaren Hardware macht.

Durch ihre modulare Bauweise ist es ebenfalls möglich, die VHM an spezielle Aufgabengebiete anzupassen, um so die optimale Lösung für ein Problem zu erhalten. Bei der Entwicklung wurde explizit darauf geachtet, dass der VHDL-Code für die VHM so allgemein wie möglich gehalten wurde, um nicht von einer speziellen Zielarchitektur, auf welche die VHM abgebildet wird, abhängig zu sein.

Der VHBC-Compiler, welcher die Konfigurationsdaten für die VHM erzeugt, ist nun in der Lage sowohl VHDL[1] als auch EDIF[2] einzulesen, zu optimieren[3], auf die zur Verfügung stehenden SubVHMs aufzuteilen und VHBC auszugeben. Er ist modular aufgebaut und einfach zu erweitern. Es besteht somit die Möglichkeit den VHBC-Compiler um neue Eingabeformate, Codemanipulationsschritte oder Ausgabeformate zu erweitern. Die Tatsache, dass der Compiler in Java realisiert wurde gewährleistet, dass er auf allen gängigen System einsatzfähig ist

Die VHM und der VHBC-Compiler wurden aufeinander abgestimmt und funktionieren reibungslos miteinander.

[1] VHDL-Code auf der SIMPRIM-Bibliothek basierend

[2] nur EDIF 2 0 0, wobei die verwendeten atomaren Komponenten einer bestimmten Namensgebung unterliegen

[3] die Optimierung umfasst momentan nur die Logikebene

6.2 Zukünftige Arbeiten und Verbesserungsvorschläge

Die geringe Anzahl an RT-Komponenten ist daran Schuld, dass die VHM im direkten Vergleich mit den in Kapitel 2 beschriebenen anderen Ansätzen deutlich weniger Rechenleistung besitzt. Es wäre somit sehr interessant, die VHM einmal mit Synthesewerkzeugen zu realisieren, welche für die Entwicklung von fester Hardware verwendet werden, um auf diese Art einen direkten Vergleich in Bezug auf Geschwindigkeit, Flächen- und Energieverbrauch ziehen zu können.

Die Funktionseinheiten der RT-Ebene sollten mit Dividieren ausgestattet werden, was momentan nur aufgrund des enormen Ressourcenverbrauchs dieser nicht der Fall ist. Weiterhin ist überlegenswert die RT-Funktionseinheiten auf Gleitkommazahlen umzustellen. Ob dies auf einem FPGA effektiv realisiert werden kann bedarf ebenfalls einer Überprüfung.

Da Logik- und RT-Ebene noch vollkommen voneinander getrennt sind wäre die Untersuchung eines Konzepts, welches das Zusammenspiel beider Ebenen beschreibt notwendig.

Die in Kapitel 3 angesprochene Vermutung bzgl. der Reduzierung des Kommunikationsaufwands sollte weiter untersucht werden. Falls diese stimmen sollte, so würde sich der Kommunikationsaufwand innerhalb der VHM drastisch verringern. Aber nicht nur die VHM würde davon profitieren. Alle Mehrprozessorsysteme mit mehr als vier Prozessoren könnten dadurch effektiver arbeiten. Die Tiefe der Programmierung und die Rekonfigurierbarkeit sind momentan auch noch Kritikpunkte.

Die Tiefe sollte auf mindestens zwei Konfigurationen angehoben werden um so der VHM eine dynamische Rekonfigurierung zu ermöglichen. Die Realisierung dürfte keinen großen Aufwand darstellen. Darauf aufbauend wäre zu überlegen, ob eine Art Hardwaretask-Scheduling sinnvoll ist.

Weiterhin wäre überlegenswert die Anzahl der Registern innerhalb einer SubVHM zu erhöhen, um dieser die Möglichkeit zu geben mehr Ergebnisse intern zwischenspeichern zu können. Der Vorteil wäre, dass dadurch MOV-Instruktionen, welche momentan berechnete Werte weiterleiten bis diese nicht mehr benötigt werden, eingespart werden können und somit den VHBC noch kompakter und effektiver machen würde, da statt der MOV-Instruktionen andere, berechnungsrelevante Instruktionen ausgeführt werden könnten.

Desweiteren sollte der komplette Compiler überarbeitet und jeder Teil auf Relevanz und Effektivität hin überprüft werden, da im Laufe der Zeit und der Änderung der Architektur der VHM einige Teile sicherlich nicht mehr notwendig oder verbesserungsfähig sind.

Eine weitere, meiner Meinung nach notwendige Maßnahme wäre die Erweiterung der Eingabeformate um EDIF 3 0 0 bzw. 4 0 0, da diese keine Zweideutigkeiten mehr erlauben und somit ein gefestigtere Basis darstellen (siehe 2.1). Ebenfalls wäre eine Optimierung der Netzliste auf RT-Ebene denkbar, indem auf ein Computer Algebra System zurückgegriffen wird, welches man über eine Programmschnittstelle ansteuert. Ob sich der Aufwand lohnt bedarf einer genaueren Analyse.

Kapitel 7

Verwendung der Software

Das folgende Kapitel beschreibt die Verwendung des VHBC-Compilers. Illustriert wird das an einigen Beispielen.

7.1 VHBC-Compiler

Der VHBC-Compiler wird über einige Parameter gesteuert. Der Aufruf der Compilers ohne Argumente liefert die folgende Ausgabe, in der die Parameter erklärt sind:

```
xaero@zerone /media/usbdisk/vhbcc $ java vhbcc

The Virtual Hardware Byte Code EDIF compiler v1.1

Takes a EDIF file and produces somewhat optimized byte code for
the Virtual Hardware Maschine. This software is part of the diploma
thesis of Thomas Siegmund

usage: vhbcc [-warn x] [-noopt] [-subVHMcount x y] [-fpwidth x] -intype TYPE file
           -warn x             sets verbose level (0-4)
           -noopt              do not optimize the netlist
           -intype TYPE        set inputtype (edif, vhdl)
           -subVHMcount x y    maximum number of SubVHMs; x for LL, y for RT; default: 4 16
           -fpwidth x          number of bits for the fixed point representation (0<x<32); default: 10
            file               name of the input file
```

Die folgenden zwei Beispiele zeigen die Anwendung des VHBC-Compilers auf zwei verschiedene Implementierungen einer 2-Bit Volladdierers. Gut zu sehen ist das Optimierungspotential des Compilers bei dem von Synopsys generierten Volladdierer (zweites Beispiel).

Es fällt auf, dass beide Repräsentationen nach der Übersetzung die gleiche Länge des kritischen Pfades und nahezu die Gleiche Anzahl an Instruktionen haben.

Um den VHBC im EDK nutzen zu können wurde ein weiteres kleines Programm geschrieben, welches aus dem Bytecode einen String von Hex-Werten erstellt. Dem Programm wird nur der vom VHBC-Compiler erzeugte Bytecode übergeben. Die Ausgabe schreibt das Programm in eine Datei, welche den gleichen Namen hat wie der Bytecode, nur das vor der Dateiendung noch ein `_ascii` eingeschoben ist.

Den von dem Programm `vhbc_to_vhm` erzeugte String kopiert man im EDK in das die Konfiguration ausführende C-Programm.

```
xaero@zerone /media/usbdisk/vhbcc $ java vhbcc -intype edif test/My2BitAdder_mentor.edf
Process 'test/My2BitAdder_mentor.edf'
Process file...DONE
Process external signals...DONE
Generate components...DONE
Stream closed...DONE
Start optimization: 15 -> 13
const_ll_0 : 1
and2 : 4
nimp2: 0
xor2 : 4
or2  : 4
nor2 : 0
eq2  : 0
inv  : 0
imp2 : 0
nand2: 0
const_ll_1  : 0
mul  : 0
add  : 0
sub  : 0
mux  : 0
comp  : 0
remove unnecessary registers...DONE
There is/are 0 flip-flop(s)
1/1
Event["
Coding Information
        Format Version : 1.1
        Number of SubVHM's LL/RT: 1/0
        Number of input ports LL/RT : 4/0
        Number of output ports LL/RT: 3/0
        Number of constants : 0
        Length of critical path LL/RT : 6/6

" sent from  at 5:55:12.26 mem :365688 of 2031616]
```

Abbildung 7.1: Ausgabe des Compilers bei der Übersetzung eines 2-Bit Volladdierers, welcher von Entwicklungswerkzeugen von Mentor Graphics erstellt wurde.

```
xaero@zerone /media/usbdisk/vhbcc $ java vhbcc -intype edif test/My2BitAdder_synopsys.edf

Process 'test/My2BitAdder_synopsys.edf'
Process file...DONE
Process external signals...DONE
Generate components...DONE
Stream closed...DONE
Start optimization: 43 -> 14
const_ll_0 : 0
and2 : 6
nimp2: 0
xor2 : 2
or2  : 4
nor2 : 1
eq2  : 0
inv  : 1
imp2 : 0
nand2: 0
const_ll_1  : 0
mul  : 0
add  : 0
sub  : 0
mux  : 0
comp  : 0
remove unnecessary registers...DONE
There is/are 0 flip-flop(s)
1/1
Event["
Coding Information
        Format Version : 1.1
        Number of SubVHM's LL/RT: 1/0
        Number of input ports LL/RT : 4/0
        Number of output ports LL/RT: 3/0
        Number of constants : 0
        Length of critical path LL/RT : 6/6

" sent from  at 5:49:57.834 mem :433936 of 2031616]
```

Abbildung 7.2: Ausgabe des Compilers bei der Übersetzung eines 2-Bit Volladdierers, welcher von Entwicklungswerkzeugen von Synopsys erstellt wurde. Durch die Optimierung wird die Schaltung von 43 auf 14 Instruktionen verkleinert.

```
xaero@zerone /media/usbdisk/vhbcc $ java vhbc_to_vhm test/My2BitAdder_mentor.vhm

const char vhm_config_data[504] = {
               "56484243116308040060 3c0000000000000000000000000000000000000000000000000000000000000000"\
               "0000000000000048c0000000000000000000000000000000000000000000000000000000000000000000000"\
               "0000000000000000000000000000004611102c22b0886c4b088ef73842c22b088eb53948676b088e5"\
               "23a52c22b088e52b08ac22b08ac226201c407184e316156c22b088e73b089e90b08ac226b06c22b0"\
               "8ac2261d5e70b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac"\
               "22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b08ac22b"\
               "08ac22b08ac22b0880000000"};
```

Abbildung 7.3: Aufruf des Programms `vhbc_to_vhm` und der von ihm generierte String.

Anhang A

CD

Die CD beinhaltet alle Quelle, sowohl die des Compilers als auch die der VHM (im Verzeichnis Sourcen), Materialien welche zu Erstellung der Diplomarbeit verwendet wurden (im Verzeichnis Diploarbeitsmaterial) und ein komplettes System für das EDK, angepasst an das XUP-Board von Xilinx, in welchem die VHM bereits eingebunden ist.

Literaturverzeichnis

[1] S.LANGE:
Design and Implementation of a Virtual Hardware Machine, Diplomarbeit, 2002
http://lips.informatik.uni-leipzig.de/pub/2002-59

[2] N. BIERWISCH:
Realisierung der virtuellen Hardware-Maschine in VHDL, Diplomarbeit, 2004
http://lips.informatik.uni-leipzig.de/pub/2003-23

[3] PACT INFORMATIONSTECHNOLOGIE GMBH
http://pactcorp.com

[4] V. BAUMGARTE, F. MAY, A. NÜCKEL, M. VORBACH, M. WEINHARDT
PACT XPP - A Self-Rekonfigurable Data Processing Architekture, PACT Informationstechnologie GmbH
http://www.pactcorp.com/xneu/download/ersa01.pdf

[5] PACT INFORMATIONSTECHNOLOGIE GMBH
The XPP White Paper (Release 2.1)
http://www.pactcorp.com/xneu/download/xpp_white_paper.pdf

[6] QUICKSILVER TECHNOLOGY
HTTP://WWW.QSTECH.COM

[7] BOB PLUNKETT, JOHN WATSON
Adapt2400 ACM Architecture Overview, Quicksilver Technology,
http://www.qstech.com/pdfs/Adapt2400_Whitepaper_0404.pdf

[8] TAIWAN SEMICONDUCTOR MANUFACTURING COMPANY, LIMITED
HTTP://WWW.TSMC.COM

[9] UNIVERSITY OF CALIFORNIA, IRVINE
http://www.uci.edu

[10] THE HENRY SAMUELI SCHOOL OF ENGINEERING
http://www.eng.uci.edu

[11] MORPHOSYS, UNIVERSITY OF CALIFORNIA, IRVINE
http://www.eng.uci.edu/morphosys

[12] HARTEJ SINGH, MING-HAU LEE, GUANGMING LU, FADI J. KURDAHI, NADER BAGHERZADEH, University of California, Irvine
and
ELISEU M. C. FILHO, Federal University of Rio de Janeiro, Brazil:
MorphoSys: An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications, University of California, Irvine
http://www.eng.uci.edu/morphosys/docs/ieee.pdf

[13] XILINX, INC.
http://www.xilinx.com

[14] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet
http://direct.xilinx.com/bvdocs/publications/ds083.pdf

[15] ALTERA, INC.
http://www.altera.com

[16] SUN MICROSYSTEMS
http://java.sun.com

[17] KEBSCHULL, UDO (PROF. DR.)
http://www.ti-leipzig.de/person/uk.html

[18] EDIF (ELECTRONIC DESIGN INTERCHANGE FORMAT)
Die Internetseite wurde nach der Auflösung geschlossen.

[19] JHDL (JAVA HDL)
http://www.jhdl.org

[20] CONFIGURABLE COMPUTING LABORATORY DER BRIGHAM YOUNG UNIVERSITY
HTTP://SPLISH.EE.BYU.EDU

[21] BRIGHAM YOUNG UNIVERSITY
http://www.byu.edu

[22] STEVEN M. RUBIN
http://www.rulabinsky.com/steve/index.html

[23] WIKIPEDIA ZUM THEMA EDIF
http://de.wikipedia.org/wiki/EDIF

[24] EDIF TECHNICAL CENTRE
http://edif-tc.cs.man.ac.uk

[25] IHS
http://global.ihs.com

Erklärung

Hiermit erkläre ich, Thomas Siegmund, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 13. Juli 2006

Thomas Siegmund